

AD-A139 240

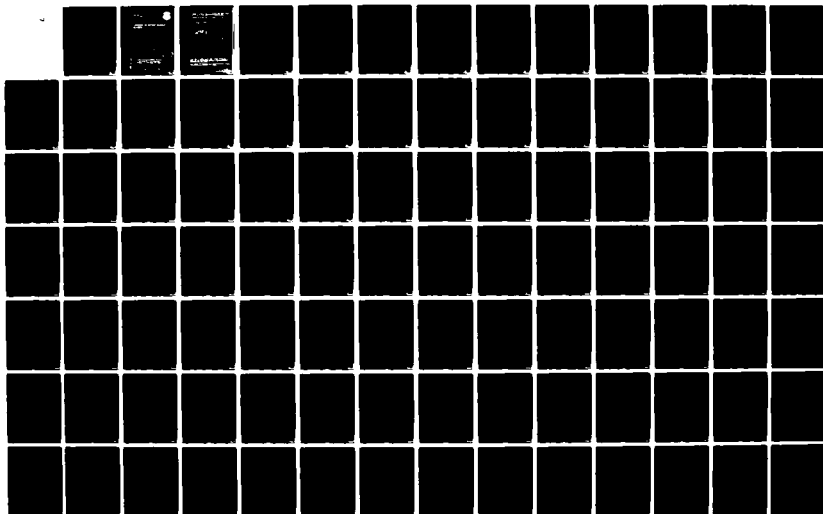
A GUIDEBOOK FOR SOFTWARE RELIABILITY ASSESSMENT(U)
SYRACUSE UNIV NY A L GOEL AUG 83 RADC-TR-83-176
F30602-81-C-0169

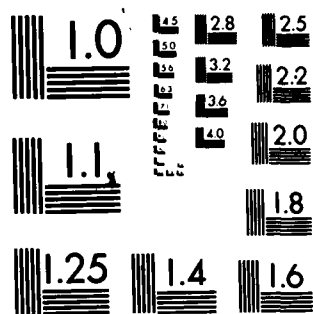
1/3

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A139240

RADC-TR-62-176
Final Technical Report
August 1962

A GUIDEBOOK FOR SOFTWARE RELIABILITY ASSESSMENT

Syracuse University

Amrit L. Goel

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC
ELECTE

MAR 29 1964

FILE COPY

RESEARCH AND DEVELOPMENT CENTER
FOR POLICE AND FIRE DEPARTMENTS
Syracuse University, Syracuse, NY 13210

James H. Brown

1952-1953

[Signature]

James H. Brown, Director, Control Division

FOR THE COMMISSIONER

[Signature]

JOHN F. BROWN
Acting Chief, Finance Division

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-83-176	2. GOVT ACCESSION NO. AD-A139240	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A GUIDEBOOK FOR SOFTWARE RELIABILITY ASSESSMENT		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report January 1982 - April 1983
7. AUTHOR(s) Amrit L. Goel		6. PERFORMING ORG. REPORT NUMBER N/A
9. PERFORMING ORGANIZATION NAME AND ADDRESS Syracuse University Syracuse NY 13210		8. CONTRACT OR GRANT NUMBER(s) F30602-81-C-0169
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COEE) Griffiss AFB NY 13441		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 558120P1
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same		12. REPORT DATE August 1983
		13. NUMBER OF PAGES 240
		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same		
18. SUPPLEMENTARY NOTES RADC Project Engineer: Joseph P. Cavano (COEE)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Software Reliability Software Reliability Models Software Reliability Assessment		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The purpose of this guidebook is to provide state-of-the-art information about the selection and use of existing software reliability models. Towards this objective, we have presented a brief summary of the available models backed by a detailed discussion of most of the models in the appendices. One of the difficulties in choosing a model is to find a match between		

DD FORM 1473 EDITION OF 1 NOV 68 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

the testing environment and a class of models. To help a user in this process, we have presented a detailed discussion of most of the assumptions that characterize the various software reliability models.

The process of developing a model has been explained in detail and illustrated via numerical examples.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

ACKNOWLEDGEMENTS

The idea of preparing a guidebook like this was originally conceived together with Alan Sukert who, due to a change of jobs from RADC to General Electric, was unable to be an active participant in its preparation. However, he continued to help me whenever needed and I owe him special thanks for his efforts.

Ashish Deb and Peter Valdes of Syracuse University provided significant help in all phases of this work. I have also benefitted greatly from discussions with Paul Moranda of McDonnell Douglas and Victor Basili and Richard Hamlet of the University of Maryland about the assumptions and applicability of the models. Joe Cavano of RADC provided detailed comments on an earlier version which have greatly improved this report. I am very thankful to all of them for their contributions to this guidebook.

I also thank Deane Bergstrom of RADC who was the prime mover behind this task.

Finally, I would like to thank Louise Naylor for her excellent typing work.



A-1	

TABLE OF CONTENTS

	<u>Page</u>
I. INTRODUCTION	1-1
1.1 Background	1-1
1.2 Limitations and Applicability of the Guidebook	1-4
2. SOFTWARE QUALITY, TESTING AND RELIABILITY	2-1
2.1 Software Quality Problem.	2-1
2.2 Software Testing	2-7
2.3 Software Reliability	2-11
2.4 Models for Software Reliability Assessment.	2-18
3. TIMES BETWEEN FAILURES (TBF) MODELS	3-1
3.1 Jelinski and Moranda De-eutrophication Model (Model TBF1)	3-3
3.2 Schick and Wolverton Linear Model (Model TBF2).	3-5
3.2.1 Schick and Wolverton Parabolic Model (Model TBF3)	3-5
3.3 Geometric De-eutrophication Model (Model TBF 4)	3-8
3.3.1 Hybrid Geometric Poisson Model (Model TBF5)	3-8
3.4 Goel and Okumoto Imperfect Debugging Model (Model TBF6)	3-10
3.5 Littlewood-Verrall Bayesian Model (Model TBF 7)	3-11
4. FAILURE COUNT (FC) MODELS.	4-1
4.1 Goel-Okumoto Non-Homogeneous Poisson Process Model (Model FC1)	4-3
4.1.1 Schneidewind Model (Model FC2)	4-5
4.2 Goel Modified Non-Homogeneous Poisson Process Model (Model FC3)	4-6
4.3 Musa Execution Time Model (Model FC4)	4-8
4.4 Shooman Exponential Model (Model FC5)	4-9
4.5 Geometric Poisson Model (Model FC6)	4-10
4.6 Modified Jelinski-Moranda Model (Model FC7)	4-11
4.7 Modified Geometric De-Eutrophication Model (Model FC8)	4-12
4.8 Modified Schick and Wolverton Model (Model FC9)	4-13
4.9 Generalized Poisson Model (Model FC10).	4-14
4.10 IBM Binomial Model (Model FC11)	4-15
4.11 IBM Poisson Model (Model FC12)	4-17

TABLE OF CONTENTS (continued)

5.	COMBINATORIAL MODELS (FS AND IDB MODELS)	5-1
5.1	Mill's Fault Seeding (FS1) Model.	5-2
5.2	Input Domain Based (IDB) Models	5-3
5.2.1	Nelson Model (IDB1 Model).	5-3
5.2.2	Ho Model (IDB2 Model).	5-5
5.2.3	Ramamoorthy and Bastani Model (IDB3 Model)	5-7
6.	ASSUMPTIONS, LIMITATIONS, AND APPLICABILITY OF MODELS.	6-1
6.1	Assumptions and Limitations	6-3
6.1.1	Independent Times Between Failures	6-3
6.1.2	A Detected Fault is Immediately Corrected.	6-4
6.1.3	No New Faults are Introduced during the Fault Removal Process.	6-5
6.1.4	Failure Rate is Proportional to the Number of Remaining Faults	6-6
6.1.5	Failure Rate Decreases with Test Time.	6-7
6.1.6	Increasing Failure Rate Between Failures	6-8
6.1.7	Testing is Representative of the Operational Usage.	6-9
6.1.8	Reliability is a Function of the Number of Remaining Faults	6-10
6.1.9	Use of Time as Basis for Failure Rate.	6-11
6.2	Applicability of Existing Software Reliability Models.	6-12
6.2.1	Design Phase	6-12
6.2.2	Unit Testing	6-14
6.2.3	Integration Testing.	6-15
6.2.4	Acceptance Testing	6-18
6.2.5	Operational Phase.	6-18
7.	A STEP BY STEP PROCEDURE FOR SOFTWARE RELIABILITY MODELING AND ILLUSTRATIVE EXAMPLES	7-1
7.1	Step by Step Procedure for Modeling	7-2
7.2	An Example of Software Reliability Modeling	7-6
7.3	Details of Parameter Estimation for the Data of Section 7.2	7-16
7.4	Numerical Examples.	7-20
8.	CONCLUSIND REMARKS	8-1
9.	REFERENCES	9-1
APPENDICES		
A.	Software Errors: Their Causes and Classification.	A-1
B.	Basic Reliability Concepts	B-1
C.	Details of Times Between Failures (TBF) Models	C-1
D.	Details of Failure Count (FC) Models	D-1
E.	Details of Combinatorial (FS and IDB) Models	E-1
F.	Selected Software Engineering Terms.	F-1

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
3.1 A Typical Plot of the Hazard Function for Model TBF1 ($N = 100, \phi = 0.01$)	3-4
3.2 Plot of a Typical Hazard Function for Model TBF2 ($N = 150, \phi = 0.02$)	3-6
3.3 A Typical Plot of the Hazard Function for Model TBF ($D = 0.5, k = 0.95$)	3-9
4.1 A Typical Plot of the Failure Rate Function for Model FC1 ($a = 175, b = 0.05$)	4-4
4.2 A Typical Plot of the Failure Rate Function for Model FC3 ($a = 500, b = 0.015, c = 1.5$)	4-7
7.1 Flowchart for Software Failure Data Analysis and Decision Making	7-3
7.2 Plot of the Number of Failures Per Hour (SYS1). . .	7-8
7.3 Number of failures and 90% Confidence Bounds (SYS1).	7-9
7.4 Observed and Expected Number of Remaining Faults with 90% Confidence Bounds on $[E \bar{N}(t)]$. . .	7-12
7.5 Reliability and 90% Confidence Bounds (SYS1) . . .	7-13
7.6 S/W Reliability Growth with Parameter Updating . .	7-15
A.1 Functional View of Software.	A-2
A.2 Software Error	A-3
C.1 Plots of PDF and CDF for Model TBF1 ($N = 100, \phi = 0.01$)	C-4
C.2 Plots of the Reliability and MTTF for Model TBF1 ($N = 100, \phi = 0.01$)	C-6
C.3 Plots of PDF and CDF for Model TBF2 ($N = 150, \phi = 0.02$)	C-10
C.4 Plots of Reliability and MTTF for Model TBF2 ($N = 150, \phi = 0.02$)	C-12

LIST OF TABLES

<u>Table</u>	<u>Page</u>
6.1 List of Key Assumptions by Model Category . . .	6-13
7.1 Failures in One Hour (Execution Time) Intervals and Cumulative Failures	7-7
7.2 Kolmogorov-Smirnoff Test for Data Set of Table 7.1.	7-11

1. INTRODUCTION

1.1 Background

An important quality attribute of a computer system is the degree to which it can be relied upon to perform its intended function. Evaluation, prediction, and improvement of this attribute have been of concern to designers and users of computers from the early days of their evolution. Until the late sixties, attention was almost solely on the hardware related performance of the system. In the early seventies, software also became a matter of concern, primarily due to a continuing increase in the cost of software relative to hardware, in both the development and operational phases of the system.

Software is essentially an instrument for transforming a discrete set of inputs into a discrete set of outputs. Since, to a large extent, software is produced by humans, the finished product is often imperfect. It is imperfect in the sense that a discrepancy exists between what the program can do and what the user or the computing environment wants it to do. These discrepancies are called software faults.

Even if we know that software contains faults, we generally do not know their exact identity. Currently, there are two approaches for exposing software faults:

program proving and program testing. Program proving, though formal and mathematical, is still an imperfect tool for verifying program correctness. Program testing is more practical but somewhat heuristic.

Due to the imperfectness of these approaches in assuring a correct program, a metric is needed which reflects the degree of program correctness and which can be used in planning and controlling additional resources (time and money) needed for enhancing software quality. One such quantifiable metric of quality that has become popular in software engineering practice is software reliability.

A number of models have been proposed during the last ten years for assessing software reliability. However, very few efforts [ANG80, GOE83] have been undertaken to evaluate their assumptions and limitations.* Also, information about the applicability of these models during various phases of software development has been lacking.

This report presents a summary and evaluation of most of the available models for software reliability assessment. A discussion of software quality, software testing, and software reliability is provided in Section 2, and a brief description of the times between failures and

*A study sponsored by RADC is currently in progress to assess the applicability of selected models to field data from an on-going project.

failure count software reliability models is given in Sections 3 and 4, respectively. Fault seeding and input domain based models are described in Section 5. Assumptions, limitations and applicability of these models are discussed in Section 6. A step by step procedure for developing a software reliability model is given in Section 7, and some of the steps are illustrated via numerical examples.

1.2 Limitations and Applicability of the Guidebook

The purpose of this guidebook is to provide a comprehensive treatment of most of the analytical models proposed during the last ten years for software reliability assessment. As stated above, the guidebook also contains a discussion of the model assumptions, their limitations and their applicability during various phases of the software life cycle.

The state-of-the-art of software reliability modeling is akin to a moving target. It was felt, however, that the time had come to document the relevant material about the available techniques so that future efforts could be focussed on resolving the unsolved problems. This viewpoint played a key role in the treatment of the material presented here. Specifically, the inclusion of any model in the guidebook does not necessarily imply that such a model is the right one to use for software reliability assessment. A decision about the usability or otherwise of a model in a given testing situation should be based on a clear understanding of the assumptions and limitations of that model vis-a-vis the actual testing process.

The following points should be helpful in determining the limitations and applicability of this guidebook.

1. The guidebook does not recommend any particular model or a class of models. It encourages the user to understand the model assumptions and its limitations before

using it in a given situation.

2. The methodology described in the guidebook for developing a reliability model is a general one and should be usable in many situations. However, other approaches may be more appropriate in certain development environments and should be preferred.
3. Most of the models use time (execution or calendar) as a basis for studying fault occurrence processes. The entity time should be interpreted in a broader sense. Any other measure which may be more relevant in a given situation can be used in lieu of time if the model assumptions are satisfied. Some examples of such measures are lines of code tested, number of functions tested, and number of test cases executed.
4. The models described in the guidebook treat the software product as a black box at a macro level. They do not explicitly take into account the effects of development methodologies and support tools. If such considerations are of interest and concern, the models described here may not be the appropriate ones to use.
5. Most of the models apparently were developed for use during the system testing stage to estimate software reliability in the field. Many of these, however, can also be used in other development phases if the underlying assumptions are satisfied.

2. SOFTWARE QUALITY, TESTING, AND RELIABILITY

2.1 Software Quality Problem

The importance of software quality in determining the performance of a computer system has been well recognized during the last decade. Currently, the low quality of software is the limiting factor in achieving overall system quality. Among the reasons for low software quality are the need for extensive human involvement in software development and the uncertainty and complexity of system applications.

Software quality assessment is different from hardware quality assessment. The latter is primarily concerned with the accuracy of fabricating (copying) the hardware design. Controlling the quality of fabricated designs is known in engineering as Quality Control. This aspect of quality has been traditionally quantified by using statistical techniques. The quality of software, on the other hand, is determined primarily by the quality of the design. The development and implementation processes that go into the production of software cannot be easily quantified at present.

The software development process can be divided roughly into five phases:

- (1) Requirements analysis. Requirements analysis involves understanding the user requirements

and expressing these requirements in the form of formal or informal specifications.

- (2) Design. The design phase further refines the specifications to come up with computer compatible specifications.
- (3) Programming. Programming is the act of implementing the specifications in a specified computing environment. In general, programming involves coding; however, pre-packaged modules or program generations may be available to satisfy the specifications.
- (4) Verification and validation. Verification and validation is the process of convincing the developer and the user that the program meets the specifications. Two complementary techniques are used to verify and validate the correctness of a program: (a) proving, and (b) testing. Program proving involves constructing a finite sequence of logical statements ending in the statement to be proved (usually the output specification). Program testing, on the other hand, involves the symbolic or physical execution of a set of test cases with the intent of exposing embedded faults, if any, in the program. Because of the complexity of current software applications, software verification is the most tiring, expensive and unpredictable phase of software develop-

ment. Roughly, 40%-50% of total development effort is spent on software verification and validation.

- (5) Operation and maintenance. Operation and maintenance refer to the actual usage of the software and all activities pertaining to correction of errors during operation, upgrading to maintain compatibility with the changing environment, and the introduction of minor or major improvements in the software.

To minimize and possibly eliminate the problem of low quality software, software engineers emulated the statistical standards of traditional engineering quality control. A number of software metrics and models have been proposed to provide numerical measures of software quality. Some metrics assign numerical weights to the number of operands/operators, number of branches, module sizes, number of GO TO statements, etc. and use these as measures of software quality. Sometimes, a mathematical formula is used to weigh these measures and obtain a single measure of quality. Some approaches in this area involve correlating or regressing software metrics to quality measures such as the number of observed errors. For example, studies have been undertaken to analyze correlations between the number of errors detected, size of software, number of operators/operands, etc. [BAS83].

A criticism of this static approach is the fact that the acceptability and quality of the implemented software design cannot be solely determined by the number of operators/operands, number of unconditional branches, module size or the like. Software quality assessment must be more than a bean-counting function. Assessment of software quality should, theoretically, take into account all the processes that go into the design, programming, verification and validation of the software.

Software errors and their impact on quality*

The imperfectness of existing design, coding, verification and validation techniques, as well as imperfectness of humans, causes omission and commission of errors in the software. It is the nature and number of these errors that effect the quality of the produced software.

A software error is any discrepancy between what the software can do versus what the user or the computing environment (i.e., physical machine, operating system, compilers, etc.) wants it to do. It can be as trivial as a syntax or semantic error or as complex as a run-time, specification, or performance error. Run-time errors, occurring during actual program execution, may be in the form of domain, computational (logic), or non-termination

*The terms error and fault are sometimes used interchangeably in this report. Definitions of selected terms in software engineering are given in Appendix F.

errors. Specification errors, occurring as a result of discrepancies between user requirements and the statement of specifications, can be due to incomplete, inconsistent, or ambiguous specifications. Performance errors exist whenever a discrepancy exists between the actual performance (efficiency) of the program and its desired or specified performance. (For a classification and description of software errors, see Appendix A.)

Software errors detected late in the software development process are much more expensive to eliminate than errors discovered early in the development process. It is, therefore, desirable that software errors which directly affect the quality of software, should be prevented and exposed as early in the software development process as possible. Basically, there are two complimentary approaches for achieving this objective:

- (1) Reduce or prevent the number of committed software errors through better requirement analysis, specification and design techniques, and better programming/implementation disciplines.
- (2) Increase error exposure through improved verification and validation techniques. Verification and validation (e.g., testing) should be distributed throughout the development process to assure early exposure of errors. Monitoring of software quality should be done at each phase of software development.

It is well recognized that the quality of software can be built into it during development by using effective software design methods for coding disciplines. A good design almost always results in good implementation, but a poor design almost always does not. A number of design philosophies are becoming popular in practice: (1) functional decomposition design, e.g., functional decomposition method, softech design method, top-down design method [BER81, PET77, GRI78]; (2) data flow design, e.g. Constantine's structured design method, Myers' composite design method [BER81, MYE75]. and (3) data structure design, e.g. Warnier/Orr Logical Construction of Programs method, Jackson design method [BER81, ORR78, JAC76].

Structured programming, or programming using restricted constructs such as IF THEN (ELSE), DO WHILE, REPEAT UNTIL, etc. and avoidance of GO TO statements, is becoming popular as an effective coding discipline.

Maximum and early exposure of errors during the software development process requires using reliable proving and testing strategies at the right phase of the development cycle. However, one should be aware that the program proving and program testing remain imperfect tools for verifying program correctness. Neither one of them can, in practice, guarantee program correctness. Proving and testing should not be viewed as competing tools; they are, in fact, complementary methods for decreasing the likelihood of program failure [GOO77].

2.2 Software Testing

Software testing is largely a heuristic process. There have been attempts to formulate a theoretical foundation of software testing (see [G0077]). The basic problem of testing is to find a test selection rule that constitutes a reliable test. A reliable test is a test which is sufficient for verifying the program's correctness. To make the testing effort practical -- requiring less effort and cost to use -- one would, naturally, pick only a small subset of the input domain that hopefully would reveal all the errors in a program. Howden [HOW76] argued that this is impossible since one can always create a program which can defeat the test. He contented himself with finding a testing strategy that is reliable for a subclass of programs. It is well known that testing can only reveal the presence of errors, never their absence.

Deterministic testing techniques can be further grouped into the structure dependent and the structure independent techniques. Structure dependent or white box testing views a program as a directed flow graph in which a node represents a set of statements and an edge is the control flow of the program. Test cases are generated based on the program's flow graph. Structure independent testing or black box testing generates test cases based on the specifications (functions) of the program. Testing the specified functions (functional testing) of a program can be a partially

random process if test cases are generated randomly for each function in the program. It is not a purely random process since each function has to be tested. On the other hand, if the test cases for each function are generated based on the program structure, the testing process becomes deterministic.

A connection exists between structure independent and structure dependent testing. The program paths from the program flow graph are really nothing but partitions of the input domain. All paths leading to an output or combination of outputs are nothing but equivalent partitions which will cause the execution of the path.

Popular examples for structure dependent testing are path testing, symbolic testing, domain testing, and mutation testing. For structure independent testing, we have equivalence partitioning, boundary value testing, and cause-effect graphing. Path testing requires that all edge-edge (path) transitions in the program flow graph be executed. A less stringent test criterion is branch testing which only requires coverage of all the edges (branches) of the graph. A further less stringent testing criterion is statement testing which merely requires coverage of all the nodes of the program graph.

Symbolic execution utilizes symbolic input to come up with outputs which are symbolic expressions of the inputs. Domain testing is currently limited to linear programs because of the difficulty, in general, in deriving

test cases from the path predicates of the program. The path predicate of a path is the condition that a set of input data has to satisfy in order for a path to be traversed at run-time. Mutation testing makes a series of minor changes to the program, creating a set of programs known as mutations. Test data is generated to cause every inequivalent mutation to give incorrect results on some input. Equivalence partitioning partitions the specified input domain into a finite number of equivalence classes. Test cases are then derived for each class pretending that the test case is representative of all members of that class. Boundary-value testing is similar to equivalence partitioning but requires that one or more elements from an equivalence class be selected such that each edge of the equivalence class is subjected to a test. Cause-effect graphing uses a cause-effect graph to generate test cases. It determines combinations of input conditions which map to a specific output condition.

A practical approach for verifying the correctness of real world software would be a combination of testing and proving coupled with the aid of software tools such as: debugging packages, program instrumentation, software/hardware monitors, simulators, compilers, link editors, static and dynamic analyzers, regression test systems, test case generators. Currently, this is the only viable approach for maximizing the exposure of embedded errors

in software. An effective software verification plan should cover the whole development process so that errors are exposed and corrected as early as possible.

2.3 Software Reliability

A quantifiable measure of quality that has become popular in software engineering practice is software reliability. It was necessitated by the inability of existing software verification and validation techniques to guarantee correct software.

There are a number of views as to what software reliability is and how it should be quantified. Some people believe that this measure should be binary in nature so that an incorrect program would have zero reliability while a perfect program would have a reliability one. This view parallels that of program proving whereby the program is either correct or incorrect. Others, however, feel that software reliability should be defined as the relative frequency (or percentage) of the times that the program works as intended by the user. This view is similar to that taken in testing where a percentage of the successful cases is used as a measure of program quality.

According to the latter viewpoint, software reliability is a probabilistic measure and is defined as the probability that a software error which causes discrepancies from specified requirements in a specified environment does not lead to a failure during a specified exposure period. Note that the probabilistic nature of this measure

is due to the uncertainty in the usage of the various software functions. Such discrepancies are also known as software faults. The specified requirements refer to the functional requirements desired by the user. Specified environment means that the software need be correct only for its specified inputs and specified computing environment. The specified exposure period may mean: (1) a single run or a number of runs; or (2) time unit expressed as CPU time units or calendar time. In simple terms, if a user executes a software product several times (according to the distribution of his needs) and 95% of the time the software provided him with acceptable (correct) results, then the software is said to be 95% reliable.

A more precise definition of software reliability which captures the points mentioned above follows [MOR83] Let F be a class of faults, defined arbitrarily, and T be a measure of relevant time, the units of which are dictated by the application at hand. Then the reliability of the software package with respect to the class of faults F and with respect to the metric T , is the probability that no error of the class occurs during the execution of the program for a prespecified period of relevant time.

Assessment of software reliability can be a non-trivial process. The reasons are as follows:

- (1) Most software is large and complex. Embedded faults may not be easily detectable by existing verification and validation techniques.

- (2) Users are not always 100 percent certain about their requirements. User input and functional distributions are not easily predictable.
- (3) Resources (time and money) allocated for software development are always limited; hence, the developer may not have enough time and money to test for all possible user inputs.

Granting that software reliability can be measured despite the above obstacles, a logical question is what purpose does it serve. Software reliability is a useful measure in planning and controlling resources (time and money) during the software development process so that high quality software can be developed. It is also a useful measure for giving the user confidence about software performance. Planning and controlling testing resources via the software reliability measure can be done by balancing the additional cost of testing and the corresponding improvement in software reliability. As more and more errors are exposed by the testing and verification process, the additional cost of exposing the remaining errors generally rises very quickly. Thus, there is a point beyond which continuation of testing to further improve the reliability of software can be justified only if such improvement is cost effective. An objective measure like software reliability can be used to study such a trade-off.

The current approaches for measuring software reliability, basically, parallel those of hardware reliability

assessment. However, appropriate modifications have been made before extending the hardware theory to software to account for the inherent differences between software and hardware. Hardware exhibits mixtures of decreasing and increasing failure rates. The decreasing failure rate is due to the fact that, as time on the hardware system accumulates, failures (most probably from design errors) are encountered and the errors fixed. The increasing failure rate is due primarily to hardware component wearout or aging. There is no such thing as wearout in software. It is true that software may become obsolete because of changes in the user and computing environment, but once we modify the software to reflect these changes, we no longer talk of the same software but of an enhanced or modified version. Like hardware, software exhibits a decreasing failure rate (improvement in quality) as the usage time on the system accumulates and errors (due to design and coding) are fixed. Thus, a hardware-based approach to software reliability assessment can be used only in appropriate environments.

It should be noted that an assessed value of the software reliability measure is always relative to a given user environment. Two users exercising two different sets of paths in the same software may have different values of the reliability of software.

A number of analytical approaches have been developed

to address the problem of software reliability assessment. These approaches are based mainly on the failure history of the software. They may be divided into time-dependent and time-independent approaches. The time dependent approach is based on either times between software failures or on failure counts in specified time intervals. The time independent approach uses either fault seeding methods or input domain analysis.

In the time dependent approach, the times between failures or the number of failures observed in a sequence of test time intervals are used to estimate the shape of the hypothesized failure (hazard) rate function. From the estimated failure rate function, one can estimate the number of faults remaining in the software, mean-time-to-failure (MTTF), and software reliability.

In the fault-seeding approach, a known number of faults is seeded (planted) in the program. After testing, the numbers of exposed seeded and indigenous faults are counted. Using combinatorics and maximum likelihood estimation, one can then estimate the number of indigenous faults in the program and also the reliability of the program.

In the input domain based models, the procedure is to generate a set of test cases from an input (operational) distribution. The difficulty of estimating the input distribution is overcome by partitioning the input domain into

a set of equivalence classes. An equivalence class is usually associated with a program or logic path. The reliability measure is then calculated from the observed failures after symbolically or physically executing the generated test cases.

The reliability of software grows as it evolves in its life cycle. Verification and testing should be performed as early as the design stage to expose design errors. If possible, the reliability of the design should also be assessed. Currently, no tool or model is available to predict the reliability of the software as early as the design stage. Testing, to expose errors after the design phase, is usually done in stages. The first stage of testing is done at the module level by the implementing programmer. Modules are then integrated to form partial or the whole system. The system is then subjected to integration testing (also known as alpha testing). Software is then given to several "friendly users" who are willing to use the software in an operational environment. The problems encountered with the software are reported. This is known as beta testing. Finally, software is released to users and corrections are issued against it as problems are reported by users.

Essentially, this overall testing process makes software reliability a growth process. However, the reliability

of software can decrease as a result of the software correction (debugging) process. This happens when additional errors are accidentally injected into the system while removing some other errors. Switching from module testing to integration testing to beta testing may also disturb the perceived software reliability growth process. A temporary surge of exposed errors may be observed when we switch to different test strategies during the software development process. The use of better design, coding and verification techniques, coupled with effective software management techniques, would reduce the likelihood of software reliability deteriorating over its life cycle.

2.4 Models for Software Reliability Assessment

A number of models have been proposed during the last ten years for assessing software reliability. Most of them are based on the failure history of software. They can be classified according to the nature of the failure process studied as indicated below:

1) Times Between Failures (TBF) Models

In this class of models the process under study is the time between failures. The general approach is to assume that the time between, say, the $(i-1)$ st and the i th failures follows a distribution whose parameters depend on the number of faults remaining in the program during this interval. Estimates of the parameters are obtained from the observed values of times between failures. Estimates of software reliability mean time to next failure, etc. are then obtained from the appropriate equations.

2) Failure Count (FC) Models

The interest in this class of models is in the number of failures in specified time intervals rather than in the times between failures. The failure counts are assumed to follow a known stochastic process with a time dependent discrete or continuous failure rate. Parameters of the failure rate can be estimated from the

observed values of failure counts or from failure times. Estimates of software reliability, mean time to next failure, etc. can be obtained from the relevant equations.

3) Fault Seeding (FS) Models

The basic approach in this class of models is to "seed" a known number of faults in a program which is assumed to have an unknown number of indigenous faults. The program is tested and the observed numbers of exposed seeded and indigenous faults are counted. From these, an estimate of the fault content of the program prior to seeding is obtained and used to assess software reliability, etc.

4) Input Domain Based (IDB) Models

The basic approach taken here is to generate a set of test cases from an input distribution which is assumed to be representative of the operational usage of the program. Because of the difficulty in obtaining this distribution, the input domain is partitioned into a set of equivalence classes, each of which is usually associated with a program path. An estimate of program reliability is obtained from the failures observed during physical or symbolic execution of the test cases sampled from the input domain.

Another classification of the models in (1) and (2) above can be based on the inference viewpoint, classical or Bayesian. However, most of the work has been along classical lines.

3. TIMES BETWEEN FAILURES (TBF) MODELS

This is one of the earliest classes of models proposed for software reliability. When interest is in modeling times between failures, it is expected that the successive failure times will get longer as faults are removed from the software system. For a given set of observed values this may not be exactly so due to the fact that failure times are random variables and observed values are subject to statistical fluctuations.

A number of models have been proposed to describe such failures in a software system. Let a random variable T_i denote the time between the $(i-1)$ st and the i th failures. Basically, the models assume that T_i follows a known distribution whose parameters depend on the number of faults remaining in the system after the $(i-1)$ st failure. The assumed distribution is supposed to reflect the improvement in software quality as faults are detected and removed from the system.

Various models for the times between failures phenomenon are described in the following subsections. A detailed description of selected models is given in Appendix C.

Note that the models described below differ primarily in their treatment of the nature of the hazard function associated with the successive software failures.

The hazard function (also known as hazard rate or failure rate) $z(t)$ is defined as the conditional probability that a fault is exposed in the interval t to $t + \Delta t$, given that the fault did not occur prior to time t . The reliability function $R(t)$ is the probability that no faults will occur from time zero to time t . Also, the functions $z(t)$ and $R(t)$ are related in the following form:

$$z(t) = [-dR(t)/dt]/R(t)$$

or

$$R(t) = \exp\left(-\int_0^t z(x)dx\right)$$

Also, mean-time-to-failure (MTTF) = $1/z(t)$.

Estimation of reliability, once the hazard function $z(t)$ is known is thus straightforward.

For details of hazard function and other relevant reliability concepts, see Appendix B.

3.1 Jelinski and Moranda De-eutrophication Model (Model TBF1)

This is one of the earliest and probably the most commonly used models for assessing software reliability [JEL72]. It assumes that there are N software faults at the start of testing, each is independent of others and is equally likely to cause a failure during testing. A detected fault is removed with certainty in negligible time and no new faults are introduced during the debugging process. The software failure rate or the hazard function at any time is assumed to be proportional to the current fault content of the tested program. In other words, the hazard function during t_i , the time between the $(i-1)$ st and i th failures is given by

$$Z(t_i) = \phi[N - (i-1)],$$

where ϕ is a proportionality constant.

Note that this hazard function is constant between failures but decreases in steps of size ϕ following the removal of each defect.

A typical plot of the hazard function for $N = 100$, $\phi = .01$ is shown in Figure 3.1. Details of this model are given in Appendix C-1.

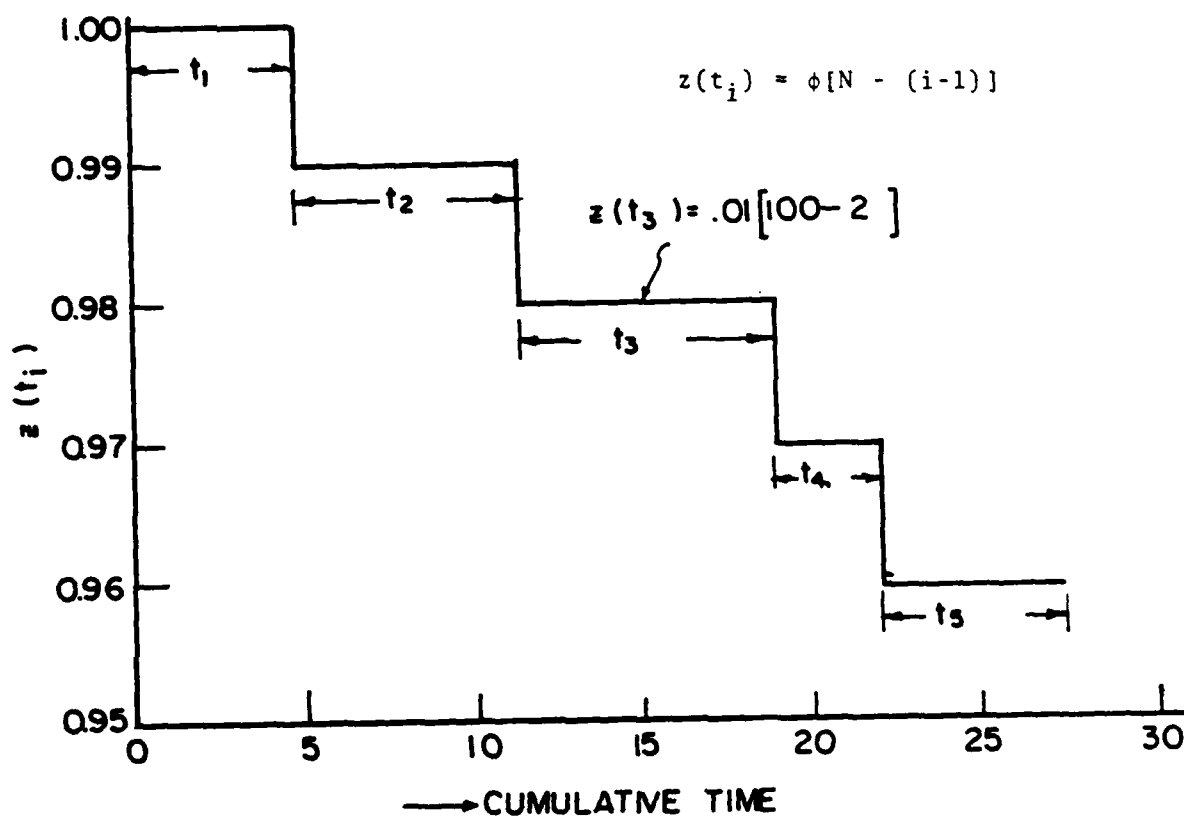


Figure 3.1 A Typical Plot of $Z(t_i)$ for Model TBF1
($N = 100$, $\phi = .01$)

3.2 Schick and Wolverson Linear Model (Model TBF2)

This model is based on the same assumptions as the TBF1 model except that the hazard function at any time is assumed to be proportional to the current fault content of the program as well as to the time elapsed since the last failure [SCH73]. Under these assumptions the hazard function $z(t_i)$ between the $(i-1)$ st and the i th failures is given by

$$z(t_i) = \phi \{N - (i-1)\} t_i$$

where ϕ is a proportionality constant and the other quantities are as defined earlier.

Note that in some papers t_i has been taken to be the cumulative time from the beginning of testing. That interpretation of t_i seems to be inconsistent with the interpretation in the original paper [SCH73].

We note that the above hazard rate is linear with time within each failure interval, returns to zero at the occurrence of a failure and increases linearly again but at a reduced slope, the decrease in slope being proportional to ϕ . This behavior for $N = 150$, $\phi = .02$ is shown in Fig. 3.2. Details of this model are given in Appendix C-2.

3.2.1 Schick and Wolverson Parabolic Model (Model TBF3)

This is a modification of Model TBF2 [SCH78] whereby the hazard function is assumed to be parabolic in test time and is given by

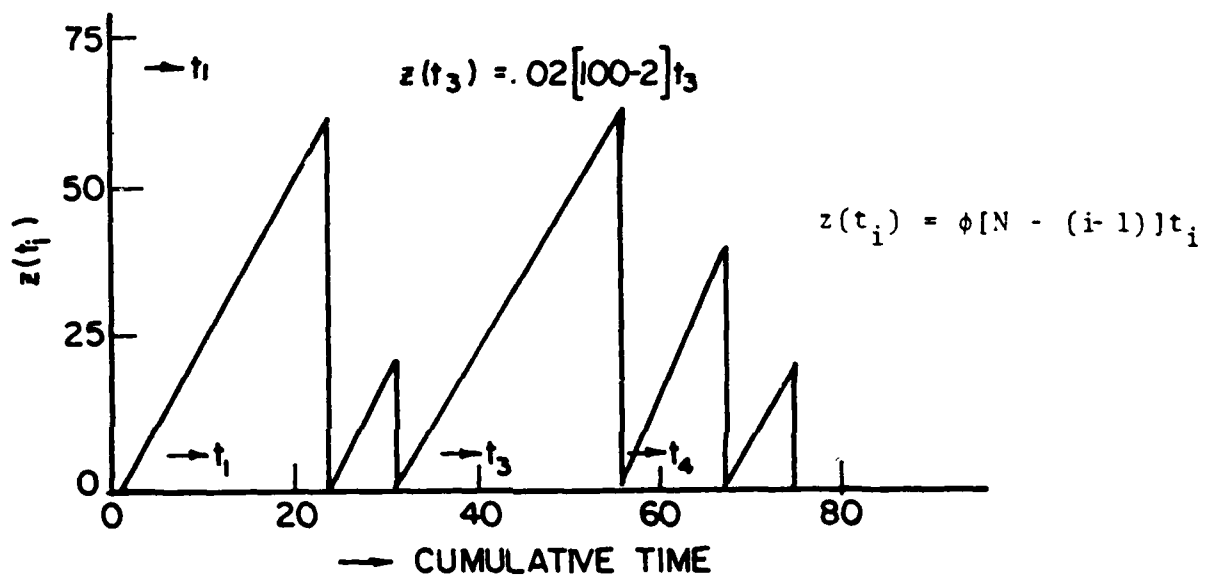


Figure 3.2. A Typical Plot for the Hazard Function for Model TBF2 ($N = 150$, $\phi = .02$)

$$z(t_i) = \phi[N - (i-1)](-at_i^2 + bt_i + c)$$

or

$$z(t_i) = \phi c[N - (i-1)] + \phi[N - (i-1)](-at_i^2 + bt_i)$$

where a,b,c are constants and the other quantities are as defined above.

This function consists of two components. The first is basically the hazard function of model TBF1. The superimposition of the second term indicates that the likelihood of a failure occurring increases rapidly as the test time accumulates within a testing interval. At failure times ($x_i = 0$), the hazard function is proportional to that of model TBF1.

3.3 Geometric De-eutrophication Model (Model TBF3)

This is a variation of the TBF1 model (3.1) and was proposed by Moranda [MOR75,MOR81] to describe testing situations where faults are not removed until the occurrence of a fatal one at which time the accumulated group of faults is removed. In such a situation, the hazard function after a restart can be assumed to be a fraction of the rate which attained when the system crashed. For this model, the hazard function during the i th testing interval is given by

$$z(t_i) = Dk^{i-1},$$

where

D is the fault detection rate during the first interval, and

k is a constant, $0 < k < 1$.

A typical plot of $z(t_i)$ for $D = 0.5$ and $k = 0.95$ is shown in Figure 3.3. Details of the model are given in Appendix C-3.

3.3.1 Hybrid Geometric Poisson Model (Model TBF4)

This model was proposed by Moranda [MOR76] as a candidate for depicting the initial segment of hardware system testing. It covers the burn in and steady state interval. It is a composite of the geometric process and a pure Poisson model. The hazard function for this model is

$$z(t_i) = Dk^{i-1} + \theta$$

where θ is the parameter of the Poisson process, and D and K are as defined above.

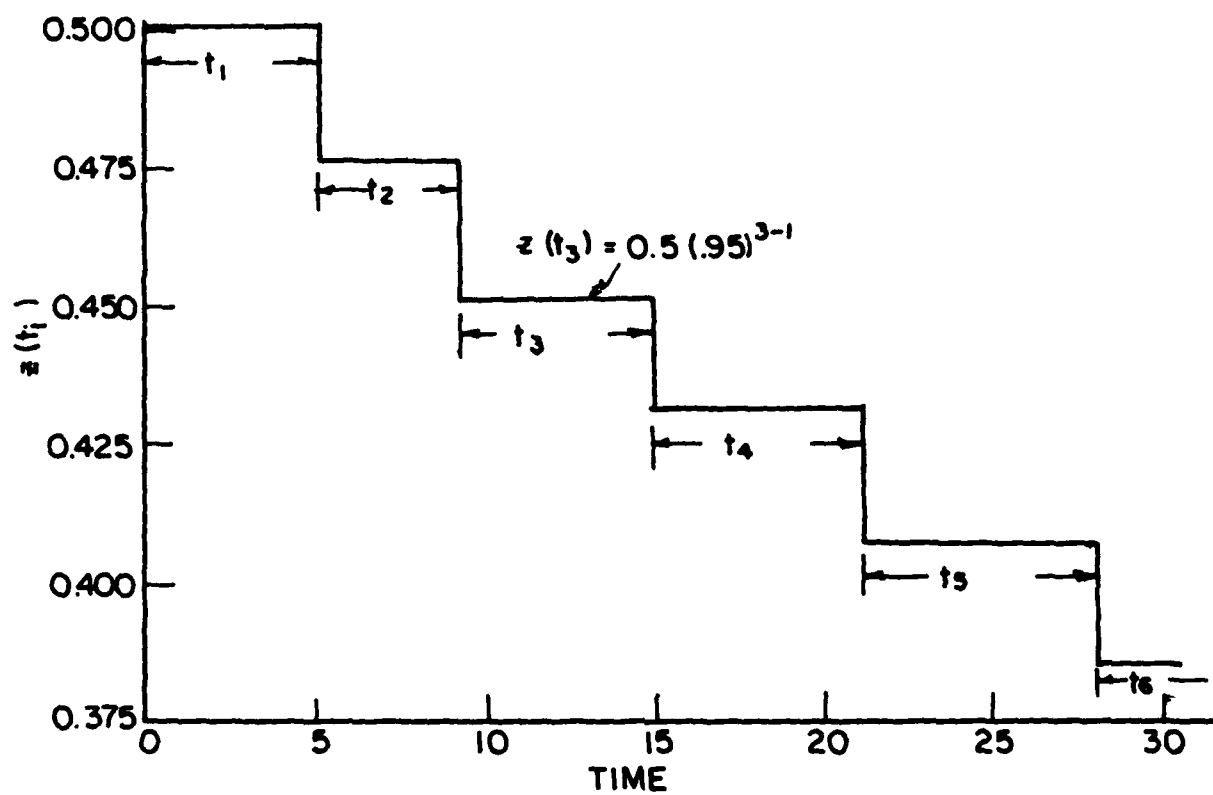


Figure 3.3 A Typical Plot of the Hazard Function for the Model TBF3 ($D = 0.5$, $k = 0.95$)

3- 4 Goel and Okumoto Imperfect Debugging Model (Model TBF5)

All of the models discussed so far assume that the faults are removed with certainty when detected. However, in practice [see MIY75,THA76] that is not always the case. To overcome this limitation, Goel and Okumoto [GOE78b,GOE78d,GOE79b] proposed an imperfect debugging model [IDM] which is basically an extension of model TBF1 [JEL72].

In this model, the number of faults in the system at time t , $X(t)$, is treated as a Markov process whose transition probabilities are governed by the probability of imperfect debugging. Times between the transitions of $X(t)$ are taken to be exponentially distributed with rates dependent on the current fault content of the system. Expressions are derived for performance measures such as the distribution of time to a completely debugged system, distribution of the number of remaining faults, and software reliability.

For this model, the hazard function during the interval between the $(i-1)$ st and the i th failures is given by

$$z(t_i) = [N - p(i-1)]\lambda.$$

where N is the initial fault content of the system,
 p is the probability of imperfect debugging,
and λ is the failure rate per fault.

3.5 Littlewood-Verrall Bayesian Model (Model TBF6)

Littlewood and Verrall [LIT73] took a different approach to the development of a model for times between failures. They argued that software reliability should not be specified in terms of the number of errors in the program. Also, they adopted a subjective approach to the treatment of failures and formulated a Bayesian model.

Specifically, the times between failures are assumed to follow an exponential distribution but the parameter of this distribution is treated as a random variable with a gamma distribution. By taking different forms for one of the parameters of this gamma distribution, it is claimed that the failure phenomena in different environments can be explained by this model.

4. FAILURE COUNT MODELS

This class of models is concerned with modelling the number of failures in given testing intervals. As faults are removed from the system, it is expected that the number of failures observed per unit time (for any reasonable form or units of the measure time) will decrease. If this is so, then the cumulative number of failures versus time curve will eventually level off. In this setup, the time intervals may be fixed a priori and the number of failures in each interval is a random variable.

Several models have been suggested to describe this failure phenomenon. The basic idea behind most of these models is that of a Poisson distribution whose parameter takes different forms for different models. It should be noted that Poisson distribution has been found to be an excellent model in many fields of application where interest is in the number of occurrences of some quantity of interest.

One of the earliest models in this category was proposed by Shooman [SH072]. Taking a somewhat similar approach, Musa [MUS75] later proposed another failure count model based on execution time. Schneidewind [SCH75] took a different approach and studied the fault counts over a series of time intervals. Goel and Okumoto [GOE79a] introduced a time dependent failure rate of the underlying Poisson process and developed

the necessary analytical details of the models. Several other models have also been proposed in this class, mostly as extensions of the corresponding time between failure models.

A brief description of those models in this category is given below. Details of selected models are presented in Appendix D.

4.1 Goel-Okumoto Non-Homogeneous Poisson Process Model (Model FC1)

In this model Goel and Okumoto [GOE79a] basically claimed that a software system is subject to failures at random times caused by faults present in the system. Letting $N(t)$ be the cumulative number of failures observed by time t , they proposed that $N(t)$ can be modelled as a non-homogeneous Poisson process (NHPP), i.e. as a Poisson process with a time dependent failure rate. Based on their study of actual failure data from many systems, they proposed the following form of the model

$$P\{N(t) = y\} = \frac{(m(t))^y}{y!} e^{-m(t)}, \quad y = 0, 1, 2, \dots$$

where

$$m(t) = a(1 - e^{-bt})$$

also

$$\lambda(t) = m'(t) = abe^{-bt}$$

In the above $m(t)$ is the expected number of failures detected by time t and $\lambda(t)$ is the failure rate. A typical plot of the $\lambda(t)$ function is shown in Figure 4.1.

In this model a is the expected number of failures to be observed eventually and b is the fault detection rate per fault. It should be noted that in this model the number of faults to be observed is treated as a random variable whose observed value depends on the test environment. This is a basic departure from most of the other models which treat the number of faults to be a fixed unknown constant.

Details of this model are given in Appendix D-1.

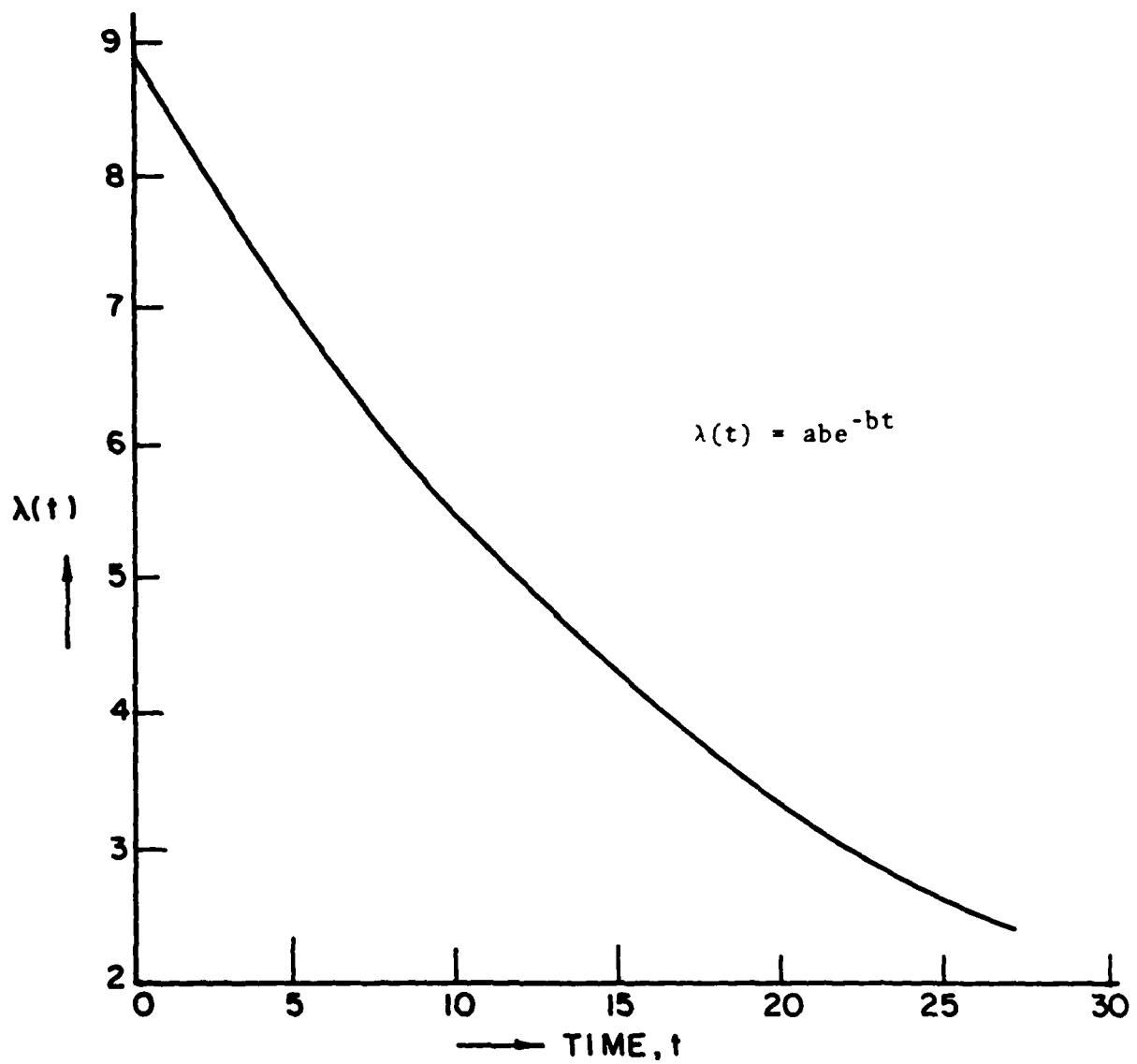


Figure 4.1 A Typical Plot for the Failure Rate Function for Model FC1 ($a = 175$, $b = 0.05$)

4.1.1 Schneidewind Model (Model FC2)

Using a different approach than described above Schneidewind [SCH75] studied the number of faults detected during a time interval and failure counts over a series of time intervals. He assumed that the failure process is a non-homogeneous Poisson process with an exponentially decaying intensity function

$$d(i) = \alpha e^{-\beta i}, \quad \alpha, \beta > 0, \quad i = 1, 2, \dots$$

4.2 Goel Modified Non-Homogeneous Poisson Process Model (Model FC3)

Most of the times between failures and failure count models assume that a software system exhibits a decreasing failure rate pattern during testing. In other words, they assume that software continues to be fault free as testing progresses.

In practice, it has been observed that in many testing situations, the failure rate (number of failures per unit time) first increases and then decreases. In order to model this increasing/decreasing failure rate process, Goel [GOE82] proposed the following modified version of the NHPP model (Model FC1).

$$P\{N(t) = y\} = \frac{(m(t))^y}{y!} e^{-m(t)}$$

where

$$m(t) = a(1 - e^{-bt^c})$$

a is expected number of faults to be eventually detected

b and c are constants that reflect the severity of testing

The failure rate is given by

$$\lambda(t) = m'(t) = abc e^{-bt^c} t^{c-1}$$

A typical plot of the $\lambda(t)$ function is shown in Figure

4.2. Details of the model are given in Appendix D-2.

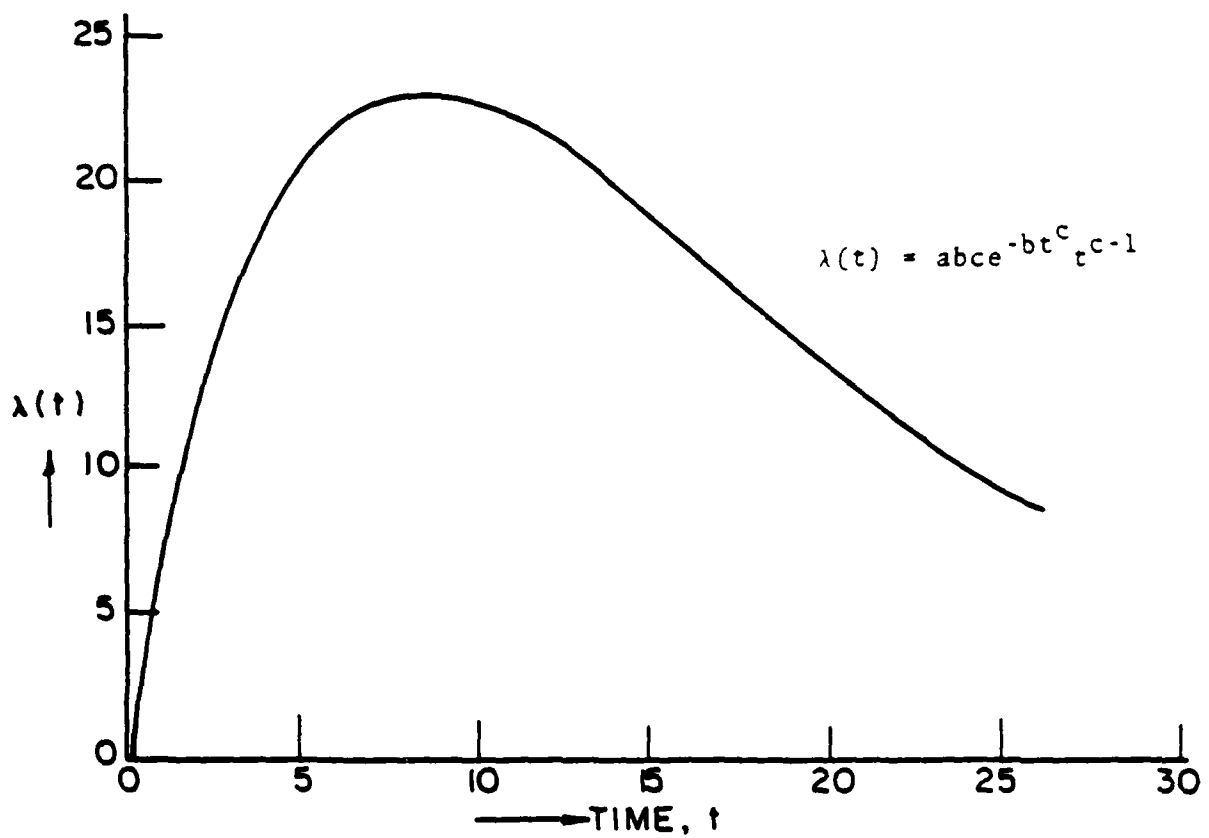


Figure 4.2 A Typical Plot of the Failure Rate Function for Model FC3 ($a = 500$, $b = 0.015$, $c = 1.5$)

4.3 Musa Execution Time Model (Model FC4)

In this model Musa [MUS75] makes assumptions that are similar to those of the model TBF1 of Jelinski and Moranda except that the process modelled is the number of failures in specified execution time intervals. The hazard function for this model is given by

$$z(\tau) = \phi f(N - n_c)$$

τ - execution time utilized in executing the program up to the present

f - linear execution frequency (average instruction execution rate divided by the number of instructions in the program)

ϕ - proportionality constant, which is a fault exposure ratio that relates fault exposure frequency to the linear execution frequency

n_c - number of faults corrected during $(0, \tau)$

One of the main features of this model is that it explicitly emphasizes the dependence of the hazard function on execution time. Musa also provides a systematic approach to converting the model so that it can be applicable to calendar time.

Details of the model are given in Appendix D-3.

4.4 Shooman Exponential Model (Model FC5)

This model is essentially similar to model TBF1 of Jelinski and Moranda. For this model the hazard function [SH072] is of the following form

$$z(t) = k\left[\frac{N}{I} - n_c(\tau)\right]$$

where

t - operating time of the system measured from its initial activation

I - total number of instructions in the program

τ - debugging time since the start of system integration

$n_c(\tau)$ - total number of faults corrected during τ , normalized with respect to I, and

k - proportionality constant

Details of this model are given in Appendix D-4.

4.5 Geometric Poisson Model (Model FC6)

Moranda [MOR75a] proposed this model to explain the failure phenomena where fault occurrence data are reported only periodically and not after each failure. He used the Poisson distribution as a description of the number of faults detected in a fixed period of time. The hazard function during the i th testing interval is given by

$$z(t_i) = \lambda K^{i-1}$$

where

λ - average number of faults occurring in the first interval

K - a constant, $0 < K < 1$

Note that this is a discrete version of the NHPP model of Goel and Okumoto (Model FC3). Details of this model are described in Appendix D-5.

4.6 Modified Jelinski-Moranda Model (Model FC7)

As the name implies, this model is a modification of model TBF1 [see SUK76]. It was proposed to explain failure phenomena where more than one fault occurs in a specified testing interval. The hazard function for the i th testing interval is given by

$$z(t_i) = \phi[N - M_{i-1}]$$

where

N - total number of faults in the system

M_{i-1} - total number of faults removed up to the
end of the previous testing interval

ϕ - proportionality constant

Details of this model are given in Appendix D-6.

4.7 Modified Geometric De-Eutrophication Model (Model FC8)

This is a modification [SUK76] of the Geometric De-Eutrophication model (Model TBF5) of Moranda. The hazard function during a testing interval is a constant whose value changes at the beginning of the next testing interval, and is given by

$$z(t_i) = Dk^{M_{i-1}}$$

where

D = fault detection rate during the first
testing interval t_1

k = a positive constant less than 1

M_{i-1} = cumulative number of faults detected up
to the end of the (i-1)st testing interval.

Details of this model are given in Appendix D-7.

4.8 Modified Schick and Wolverson Model (Model FC9)

In this model the faults are assumed to occur independently of each other. The fault occurrence rate during a testing interval is proportional to the number of faults remaining in the system at the beginning of this interval and to the total time previously spent in testing (including an 'averaged' fault search time during the current time). The expected number of faults occurring during the i th interval of length t_i is then given by

$$E[N_i] = \phi[N - M_{i-1}][T_{i-1} + \frac{t_i}{2}] t_i,$$

where

M_{i-1} is the total number of faults removed up to the end of the $(i-1)$ st interval,

t_i is the i th testing interval

T_{i-1} is the cumulative test time through the $(i-1)$ st interval, and

ϕ is a proportionality constant

A detailed description of this model is given in Appendix D-8.

4.9 Generalized Poisson Model (Model FC 10)

This is a variation of the NHPP model of Goel and Okumoto (Model FC1) and assumes a mean value function

$$m(t_i) = \phi(N - M_{i-1})t_i^\alpha$$

where

M_{i-1} - total number of faults removed up to
the end of the (i-1)st debugging interval

ϕ - constant of proportionality

α - constant used to rescale time t_i

Details of this model are given in Appendix D-9.

4.10 IBM Binomial Model (Model FC11)

In this model Brooks and Motley [BR083] consider the fault detection process in software testing as a discrete process. The software system is assumed to be developed and tested incrementally. This means that some modules of the system may be available for testing, or in and out of test, while others are not. The testing phase of the software system is assumed to consist of a number of test occasions and each test occasion is further assumed to consist of unit intervals of testing. Thus, testing effort (i.e. total number of unit intervals) in each test occasion could be different. This model also assumes that faults could be introduced into the software during the fault removal process.

This model can be applied at the module level or at the system level. If the model is applied at the module level, then the observed process is the fault occurrence process of each test occasion of the module under test. According to this model the number of faults detected during the i th test occasion in module j , n_{ij} , follows a binomial distribution with parameters \bar{N}_{ij} and q_{ij} , that is

$$P\{n_{ij} = x_{ij}\} = \binom{\bar{N}_{ij}}{x_{ij}} q_{ij}^{x_{ij}} (1 - q_{ij})^{\bar{N}_{ij} - x_{ij}}$$

where, \bar{N}_{ij} = expected number of faults remaining in module j at the beginning of the i th test occasion

n_{ij} = number of faults detected in module j
during the i th test occasion

q_{ij} = fault detection probability for the i th test
occasion from module j .

If the model is applied at the system level, then the observed process is the fault occurrence process of each test occasion of the software system. In that case, the number of faults detected during the i th test occasion of the software system follows a binomial distribution with parameters \bar{N}_i and q_i , that is

$$P\{n_i = x_i\} = \binom{\bar{N}_i}{x_i} q_i^{x_i} (1 - q_i)^{\bar{N}_i - x_i}.$$

where, \bar{N}_i = expected number of faults remaining at the beginning of the i th test occasion.

n_i = number of faults detected during the i th test occasion.

q_i = fault detection probability for the i th test occasion.

Note that $\bar{N}_i = \sum_{j \in J_i} N_{ij}$

where, J_i is the set of modules tested on occasion i .

Details of this model are given in Appendix D-10.

4.11 IBM Poisson Model (Model FC12)

The assumptions of this model are similar to those of model FC11 except that the number of faults found during a test occasion is assumed to follow a Poisson distribution.

This model also can be applied at the module or the system level. If the model is applied at the module level, then the observed process is the fault occurrence process of each test occasion of the module under test. According to this model the number of faults detected during the i th test occasion in module j follows a Poisson distribution with parameter $\bar{N}_{ij}\phi_{ij}$, that is.

$$P\{n_{ij} = x_{ij}\} = \frac{e^{-\bar{N}_{ij}\phi_{ij}} (\bar{N}_{ij}\phi_{ij})^{x_{ij}}}{x_{ij}!}$$

where, \bar{N}_{ij} = expected number of remaining faults in module j at the beginning of test occasion i .

n_{ij} = number of faults detected in module j during test occasion i

ϕ_{ij} = the proportionality factor between remaining faults and fault detection rate.

$\bar{N}_{ij}\phi_{ij}$ = expected number of faults detected in module j during test occasion i .

If the model is applied at the system level, then the observed process is the fault occurrence process of each test occasion of the software system. Then, in that case, the number

of faults detected during the i th test occasion follows a Poisson distribution with parameter $\bar{N}_i \phi_i$, that is

$$P\{n_i = x_i\} = \frac{e^{-\bar{N}_i \phi_i} (\bar{N}_i \phi_i)^{x_i}}{x_i!}$$

where, \bar{N}_i = expected number of remaining faults at the beginning of the i th test occasion.

n_i = number of faults detected during the i th test occasion.

ϕ_i = the proportionality factor between remaining faults and fault detection rate.

$\bar{N}_i \phi_i$ = expected number of faults detected in the system during test occasion i .

Note that $\bar{N}_i = \sum_{j \in J_i} \bar{N}_{ij}$

where J_i is the set of modules tested on occasion i .

Details of this model are given in Appendix D-11.

5. COMBINATORIAL MODELS (FS AND IDB MODELS)

In this section we give a brief description of time-independent models that have been proposed for assessing software reliability. As mentioned earlier, the two approaches proposed for this class of models are fault seeding and input domain analysis.

In fault seeding models a known number of faults is seeded (planted) in the program. After testing, the numbers of exposed seeded and indigenous faults are counted. Using combinatorics and maximum likelihood estimation, one can then estimate the number of indigenous faults in the program or the reliability of the software.

The basic approach in the input domain based models is to generate a set of test cases from an input (operational) distribution. Because of the difficulty in estimating the input distribution, the various models in this group partition the input domain into a set of equivalence classes. An equivalence class is usually associated with a program path. The reliability measure is calculated from the observed failures after execution (symbolic or physical) of the sampled test cases.

The fault seeding model is discussed in Section 5.1. Input domain based models are described in Section 5.2. More details about these models are given in Appendix E.

5.1 Mills Seeding Model (FS1)

A number of models have been proposed but the most popular (and most basic) is Mills' Hypergeometric model [MIL72]. This model requires that a number of known faults be randomly inserted (seeded) in the program to be tested. The program is then tested for some amount of time. The number of original indigenous faults can be estimated from the numbers of indigenous and seeded faults uncovered during the test by using the hypergeometric distribution.

The procedure adopted in this model is similar to the one used for estimating population of fish in a pond or for estimating wildlife. These models are also referred to as tagging models since a given fault is tagged as seeded or indigenous.

The relevant formulae are given in Appendix E.1.

Lipow [LIP72] modified this problem by taking into consideration the probability, q , of finding a fault (of either kind) in any test of the software. Then, for N statistically independent tests the probability of finding x_I indigenous faults and x_S seeded faults can be calculated as shown in Appendix E-1.

Basin [BAS74] suggested a two stage procedure with the use of two programmers. Estimates of the number of indigenous faults using this procedure can be obtained as shown in Appendix E-1.

5.2 Input Domain Based Models (IDB Models)

Representative models in this class are those proposed by Nelson [NEL78,BRO75], Ho [HO78] and Ramamoorthy and Bastani [RAM82].

5.2.1 Nelson Model (IDB1 Model)

The reliability of the software is measured by exposing (running) the software with a sample of n inputs. The n inputs are randomly chosen from the input domain set $E = (E_i: i = 1, N)$ where each E_i is the set of data values needed to make a run. The random sampling of n inputs is done according to a probability distribution P_i ; the set $(P_i: i = 1, N)$ is the "operational profile" or simply user input distribution. If n_e is the number of inputs that resulted in execution failures, then an unbiased estimate of software reliability \hat{R}_1 is $1 - (n_e/n)$. However, it may be the case that the test set used during the verification phase may not be representative of the expected operational usage. Brown and Lipow [BRO75] suggested an alternative formula for \hat{R} which is

$$\hat{R}_2 = 1 - \sum_{i=1}^N \frac{f_j}{n_j} P(E_j)$$

where

n_j - number of runs sampled from input subdomain E_j

f_j - number of failures observed out of n_j runs.

The main difference between Nelson's \hat{R} and Brown and Lipow's is that the former explicitly incorporates the usage distribution or the test case distribution while the latter

implicitly assumes that the accomplished testing is representative of the expected usage distribution. Both models assume prior knowledge of the operational usage distribution.

5.2.2 Ho Model (IDB2 Model)

Reliability estimation in this model proceeds by first generating the symbolic execution tree of the program. This tree characterizes all the execution paths and their associated outputs in the program. The nodes represent statements while the edges represent the state vector resulting from symbolic execution along the path from the root statement to the current statement. A procedure for generating the symbolic execution tree [HO78] is given below:

- I. The first statement is the root of the tree.
- II. If a leaf is not a STOP or RETURN statement, symbolically execute the statement corresponding to the node. If the current statement is a conditional statement, the feasibility of the branches is examined. New nodes are created for statements which are successors of the current statement. Edges, labelled with state vectors, are joined between the current node and the new node(s).
- III. Go to II.

The generated execution paths from the symbolic execution tree are proven correct or are sample tested. For a given path, say path i , if it is proven correct, then the path reliability $R_i = 1$. If path i cannot be proven correct, a random sample of N test cases is generated that will execute path i . If no failures result from the execution of the N test cases, then R_i is bounded below by $1 - C_i$ where C_i

is the confidence interval of path i . The length of C_i is a function of our given confidence coefficient α . On the other hand, if n failures are observed and the errors not corrected, then R_i is bounded below by $\frac{N-n}{N} - C_i$. If the observed n failures are corrected, then the sample testing is repeated for path i .

Finally, the software reliability estimate \hat{R} is obtained as

$$\hat{R} = \sum_{i=1}^m f_i R_i$$

where:

f_i - weighting factor of path i which corresponds to the execution frequency of path i .

m - total number of execution paths.

One difficulty with applying this approach is the large number of paths that may exist in any large size software.

5.2.3 Ramamoorthy and Bastani Model (Model IDB3)

This input domain based model estimates the reliability \hat{R} from the relation

$$\hat{R} = 1 - \hat{V}_{e_r}$$

where

\hat{V}_{e_r} - the total fault size remaining in the program.

\hat{V}_{e_r} can be determined by testing the program and locating and estimating the size of faults found. A fault has a large size if it is easily detected (i.e., if it affects many input elements). A fault has a small size if it is relatively difficult to detect. The size of a fault depends on the way test inputs are selected. Good test case selection strategies like path testing, boundary value analysis, magnify the size of a fault since they exercise error-prone constructs. The observed fault size is lower if random testing is employed. Although the model does not assume random testing (in fact, any test strategy can be employed), it offers no easy or systematic way to estimate \hat{V}_{e_r} .

6. ASSUMPTIONS, LIMITATIONS AND APPLICABILITY OF MODELS

In this section we discuss the appropriateness or otherwise of the various assumptions underlying the models of Sections 3 through 5 as well as their applicability during software development phases. Details of the assumptions have been brought out in Appendices C, D, and E along with the details of the models. Not all the assumptions discussed here are relevant to any given model but as a totality, they give a picture of the kind of limitations imposed on the use of software reliability models. The purpose of the following discussion is to focus attention on the framework within which the existing models have been developed.

It should be pointed out that the arguments presented here regarding the assumptions or the applicability of the models may not be universally acceptable. This is so because the software development process is very environment dependent. What holds true in one environment may not be true in another. Because of this, assumptions that are reasonable, e.g. during the testing of one function or system may not hold true in even a subsequent test phase of the same function or system. The ultimate decision about the appropriateness of the underlying assumptions and the applicability of the models will have to be made by the user of a model. What is presented

here should be helpful in determining whether the assumptions underlying a given model are representative of his testing environment and in deciding which model, if any, to use.

The assumptions are discussed one at a time in Section 6.1 along with a list of the models which use the assumption being discussed. Software development phases and the applicability of models in each phase are covered in Section 6.2.

6.1 Assumptions and Limitations

6.1.1 Independent Times Between Failures

This assumption requires that the times between successive failures be independent of each other. In general, this would be the case if successive test cases were independent, i.e., were chosen randomly. However, testing, especially functional testing, is not based on independent test cases, i.e., the test process in general is not a random process. The time (or the additional number of test cases) to the next failure may very well depend on the nature or time of the previous fault. If a critical fault is uncovered, the tester may decide to intensify the testing process and look for more potential critical faults. This in turn may mean shorter time to the next failure.

This assumption is used in models TBF1, TBF2, TBF3, TBF4, TBF5 and TBF6.

6.1.2 A Detected Fault Is Immediately Corrected

The models that require this assumption assume that the software system goes through a purification process as testing uncovers faults which are immediately removed. An argument can be made that this assumption is implicitly satisfied in most testing environments. When a fault is detected, the testing process in general, but not always, can proceed without removing the fault. The future failure process can then be assumed to be based on the assumption that the fault was in fact removed after its occurrence.

If, however, the fault is in the path that must be tested further, this assumption would be satisfied only if the fault is removed prior to proceeding with the remainder of the test bucket or the test cases are generated to get around it.

This assumption is used in models TBF1, TBF2, TBF3, TBF4, TBF6.

6.1.3 No New Faults are Introduced during
the Fault Removal Process

The purpose of this assumption is to ensure that the modelled failure process does have a monotonic pattern. That is, the subsequent faults are exposed from a system that has less faults than before. In general, this may not be true if the correction process follows the occurrence of every failure, because during the correction process, other paths may have been affected leading to additional faults in the system. It is generally considered to be a restrictive assumption in reliability models. The only way to satisfy this is to ensure that the correction process does not introduce new faults.

This assumption is used in all reliability models.

6.1.4 Failure Rate is Proportional to the Number of Remaining Faults

This assumption is the same as saying that each remaining fault has the same chance of being detected in a given testing interval between failures. This assumption is a reasonable one if the test cases are chosen to ensure equal probability of executing all portions of the code. However, if one portion (or a set of paths) is executed more thoroughly than another, the faults in the former are more likely to be detected than in the latter. Faults residing in the unreachable (or never tested) portion of the code will obviously have a low, or zero, probability of being detected.

This assumption is used in models TBF1, TBF2, and TBF4.

6.1.5 Failure Rate Decreases with Test Time

This assumption implies that the software gets better with testing. This seems to be a reasonable assumption and can be justified as follows. As testing proceeds, faults are detected. They are either removed before testing continues or they are not removed and the test coverage is narrowed in subsequent tests. In the former case the subsequent failure rate decreases explicitly. In the latter case, the failure rate (based upon the entire program) decreases implicitly as smaller and smaller portions of the code are subjected to testing.

This assumption is used in models FC1, FC3, FC4, and FC5.

6.1.6 Increasing Failure Rate Between Failures

This assumption implies that the likelihood of finding a fault increases as the testing time increases within a given failure interval. This would be a justifiable assumption if software were assumed to be subject to wearout within the interval. But this is not generally the case with software systems. Another situation where such an assumption might be justifiable is where testing intensity increases within the interval in the same fashion as does the failure rate.

This assumption is used in models TBF2, TBF3, and FC10.

6.1.7 Testing is Representative of the Operational Usage

The test cases are generally chosen to ensure that the functional requirements of the system are correctly met. A given user of the system, however, may not use the functions in the same proportion. Testing, then, will not reflect the operational usage. However, information about the usage profile is not easy to obtain and this assumption would be the logical one to use. If the required information is available, testing effort can be modified to be representative of the use profile.

This assumption is necessary when reliability estimate based on testing is projected into the operational phase. It is used primarily in models IDB1, IDB2, and IDB3. Most TBF and FC models would also need this assumption if they are being used to assess operational reliability.

6.1.8 Reliability is a Function of the Number of
Remaining Faults

This assumption implies that all remaining faults are equally likely to appear during the operational usage of the system. If the usage is uniform, then this is a reasonable assumption. If, however, some portions are more likely to be executed than others, the reliability of the system can be recomputed by incorporating this information. In other words, a reliability measure conditioned on usage rather than an unconditional measure would be more desirable. If, however, such information is not available, then the assumption of uniform usage is the only reliable one.

This assumption is made when reliability estimates from any model are based on the number of remaining faults.

6.1.9 Use of Time as a Basis for Failure Rate

Most models use time as a basis for determining changes in failure rate. This usage assumes that testing effort is proportional to either calendar time or execution time. Also, time is generally easy to measure and most testing records are kept in terms of time. Another argument in favor of this measure is that time tends to smooth out differences in test effort.

If testing is really not proportional to time, the models are equally valid for any other relevant measure. Some examples of such measures are lines of code tested, number of functions tested, and number of test cases executed.

6.2 Applicability of Existing Software Reliability Models

Due to the various assumptions that are required by the models described earlier (see Table 6.1), it is necessary to use caution in choosing a model for software reliability assessment. In this subsection we develop a classification scheme and suggest the classes of models that might be applicable in various phases of the software development process.

For this purpose we consider the following phases of the software development process.

1. Design
2. Unit Testing/Debugging
3. Integration Testing/Debugging
4. Acceptance Testing/Debugging
5. Operational Testing

6.2.1 Design phase

During the design phase, faults (i.e. design faults) may be corrected visually or by other formal/informal procedures. Existing software reliability models are not applicable at this stage because

- (i) test cases needed to expose faults as required by fault seeding and input domain based models do not exist,

Table 6.1

List of Key Assumptions by Model Category

1. TBF Models

- . Independent times between failures
- . Equal probability of the exposure of each fault
- . Embedded faults are independent of each other
- . Immediate fault removal, perfect fault removal, no new faults introduced during correction.

2. FC Models

- . Testing intervals are independent of each other
- . Testing during intervals is reasonably homogeneous
- . Numbers of faults detected during non-overlapping intervals are independent of each other

3. FS Models

- . Indigeneous and seeded faults have equal probabilities of being detected.

4. IDB Models

- . Input profile distribution is known
- . Random testing is used
- . Input domain can be partitioned into equivalent classes.

- (ii) failure history required by time dependent models is not available.

6.2.2 Unit testing

The typical environment during the module coding and unit testing phase is such that:

- (i) test cases generated from the module's input domain may not form a representative sample of the operational "profile" distribution,
- (ii) times between exposures of module faults are not random since the test strategy employed may not be random testing. Also, test cases are usually executed in a deterministic fashion,
- (iii) exposed faults are corrected (debugged).

Given these conditions, it seems that the fault seeding models are applicable provided it can be assumed that the indigenous and seeded errors have equal probabilities of being detected. However, a difficulty arises because the programmer is also the tester in this phase. The input domain based models seem to be applicable, except that matching the test profile distribution with the operational profile distribution is non-trivial. Due to these difficulties, such models, though applicable, may not be usable.

The time dependent models (TBF models) do not seem to be applicable in this environment since the independent times between failures assumption is seriously violated.

6.2.3 Integration testing

A typical environment during integration testing is:

- (i) modules are integrated into partial or whole systems and test cases are generated to verify the correctness of the integrated system,
- (ii) test cases may be generated randomly following an input distribution or may be generated deterministically using a reliable test strategy, the latter being probably more effective,
- (iii) exposed faults are corrected and there is a strong possibility that the removal of exposed faults may introduce new faults.

Fault seeding models are still theoretically applicable since we still have the luxury of seeding faults into the system. Input domain based models based on an explicit test profile distribution are also applicable. Again, the difficulty in applying them at this point is the large number of logic paths generated by the whole system.

If deterministic testing (i.e., boundary value analysis, path testing, etc.) is used, times between failures (TBF) models may not be appropriate because of the violation of the independence of inter-failure times assumption. Deterministic testing increases the likelihood of exposing faults and, hence, inter-failure times are no longer random. Failure count models (ex. FC1, FC2) may be applicable if sets of test cases are independent of each other, even if the tests

within a set are run deterministically. This is so because in FC1 and FC2 models the system failure rate is assumed to decrease as a result of executing a set of test cases and not at every failure.

If random testing is done according to an assumed input profile distribution, then most of the existing software reliability models are applicable. Input domain based models, if used, should utilize a test profile distribution which is statistically equivalent to the operational profile distribution. Fault seeding models are applicable likewise, since faults can be seeded and the equal probability of fault detection assumption may not be seriously violated. This is due to the random nature of the test generation process. Times between failures and failure count models are most applicable in this environment with random testing. The only question in applying these models is which specific model to use. Some people prefer to try a couple of these models for the same failure history and then choose an appropriate one. However, because of the different underlying assumptions of these models, there is still a subtle distinction as to when a specific model is most applicable. For example, for operating systems or real-time systems which are run almost continuously, the time-dependence assumption, say of the Non-homogeneous Poisson Process Model (FC1), or model FC2, are most applicable. We should, however, be aware of the fact that in most real-time systems the inputs to

the software may not be random.

If there is reason to believe that the operational input to the software is essentially uniform, then each embedded fault in the software has essentially an equal chance of being detected. Thus, times between failures models based on a constant multiple of the number of remaining faults, (e.g. Model TBF1 of Jelinski and Moranda) are more applicable. In real-world situations, the equal chance of fault detection assumption is rarely true. This is due to two main reasons:

- (i) embedded faults in the software have unequal sized, some are large while others are small,
- (ii) the input distribution may not be uniform.

If such is the case, models which assume that error occurrence rate varies with time (e.g. Model FCI) are more applicable:

In an environment where exposed errors are imperfectly debugged, a theoretically more applicable model is the Imperfect Debugging Model (Model TBF6).

In environments where imperfect debugging is assumed and additional errors are introduced during the error correction process, none of the existing software reliability models is applicable. Models of this kind are yet to be developed.

6.2.4 Acceptance testing

During acceptance testing the software is given to "friendly users". These users generate inputs (usually random following an input distribution) to verify the correctness of the software. The main differences between this environment and that of the integration testing environment are the following:

- (i) exposed faults are immediately corrected since the user may not know how to correct them,
- (ii) the user does not have the luxury of seeding faults in the program,
- (iii) the user may not even know the structural paths of the program.

It can be easily seen that the fault-seeding models are not applicable because of (ii) above. Times between failures models are also not applicable since these models assume correction of exposed errors before testing proceeds.

6.2.5 Operational phase

When the reliability of the software as perceived by the developer or the "friendly users" is already acceptable, the software is released for operational use. During the operational phase, the user inputs may not be random. This is because the user may use the same software function or path on a routine (production) basis. Inputs may also be

correlated (like in real-time systems), thus losing their randomness. Furthermore, faults are not always immediately corrected.

Software that is in operational use may be recalled by the developer so that errors exposed during the operational phase can be corrected. The usual practice is to release a newer version of the software which hopefully contains none of the exposed faults.

It appears that the failure count models could be used in this phase if the input process can be considered to be random over longer intervals of time.

7. STEP BY STEP PROCEDURE FOR SOFTWARE RELIABILITY MODELING AND ILLUSTRATIVE EXAMPLES

The purpose of this section is to describe a systematic procedure for developing a software reliability model from available failure data. The general procedure is described in Section 7.1 and illustrated via analyses of some failure data in Section 7.2. An important phase in the development of a model is the estimation of parameters, mostly done iteratively. Details of such calculations for the data set of Section 7.2 are given in Section 7.3. Estimation of parameters is further illustrated for a simple data set in Section 7.4 using the De-Eutrophication model of Jelinski and Moranda (Model TBF1) and the NHPP model of Goel and Okumoto (model FC1).

7.1 Step by Step Procedure for Modeling

The step by step procedure is shown in Figure 7.1 and described below.

Step 1: Study the failure data

Most of the models discussed in this report require the prior existence of some failure data to fit a model. The first step in developing a model is to study this data in order to gain an insight into the nature of the process being modelled. If, for example, the number of failures per unit time is increasing, it would appear that more and more functions are being added during the subsystem or system test. The data may have to be normalized before proceeding any further because most of the models assume a basically steady system. It may be that the test case severity is increasing. Again, this needs to be accounted for in the modelling process.

Step 2: Choose a Reliability Model

The next step is to choose an appropriate model based upon an understanding of the testing process and the assumptions of the various models discussed earlier. The data in Step 1 can also be used to sharpen decisions about the applicability of a class of models or of the choice of a given model.

Step 3: Obtain Estimates of Parameters of the Model

Different methods are generally required depending upon the type of available data. The most commonly used

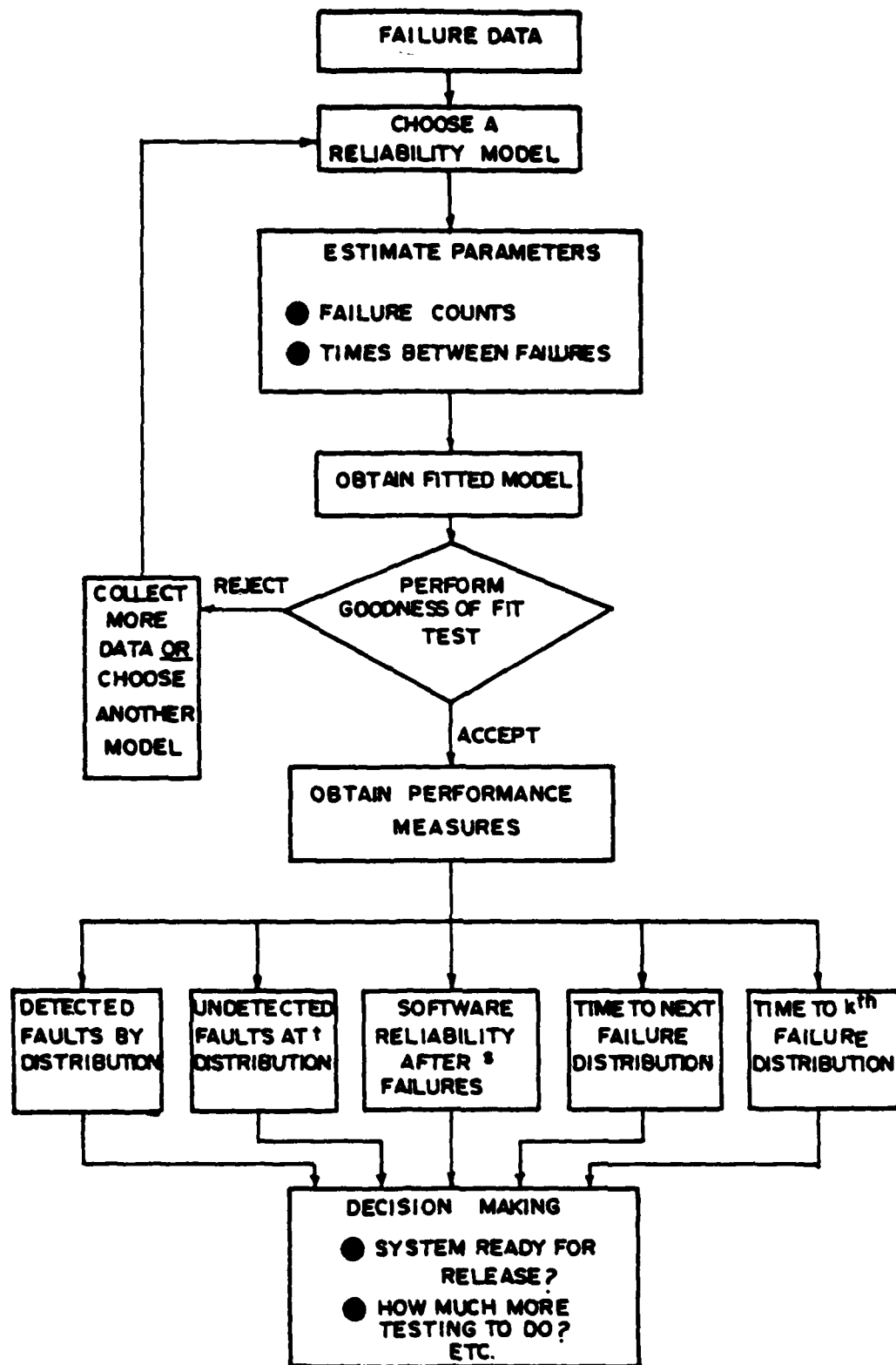


Figure 7.1 Flowchart for Software Failure Data Analysis and Decision Making.

ones are the least squares and maximum likelihood methods.

Step 4: Obtain the fitted model.

The fitted model is obtained by substituting the estimated values of the parameters in the chosen model. At this stage, we have a fitted model based on the available failure data.

Step 5: Perform goodness-of-fit test.

Before proceeding further, it is advisable to conduct the Kolmogorov-Smirnov goodness-of-fit test or some other suitable test to check the model fit.

If the model fits, we can move ahead. However, if the model does not fit, we have to collect additional data or seek a better, more appropriate model. There is no easy answer to either how much data to collect or how to look for a better model. Decisions on these issues are very much problem dependent.

Step 6: Compute confidence regions.

It is generally desirable to obtain 80%, 90%, 95% and 99% joint confidence regions for the parameters of the model to assess the uncertainty associated with their estimation. This information is helpful in the interpretation of model outputs.

Step 7: Obtain performance measures.

At this stage we can compute various quantitative measures to assess the performance of the software system. Some useful measures are shown in Figure 7.1. Confidence bounds can also be obtained for these measures to evaluate the degree of uncertainty in the computed values.

Step 8: Decision making.

The ultimate objective of developing a model is to use it for making some decisions about the software system, e.g., whether to release the system or continue testing. Such decisions are made in this step of the modelling process based on the information developed in the previous steps.

7-2 An Example of Software Reliability Modelling

In this section we employ the above step-by-step procedure to illustrate the development of a software reliability model based on actual failure data from a real-time command and control system. For the purposes of this illustration, we employ the NHPP model of Goel and Okumoto (Model FC1).

The delivered number of object instructions for the system being modeled was 21,700 and the system was developed by Bell Laboratories [MUS80].

Step 1:

The original data was given as times between failures. To overcome the independence assumption, we summarized the data into numbers of failures per hour of execution time. The data are shown in Table 7.1 and plotted in Fig. 7.2. A plot of $N(t)$, the cumulative number of failures is shown in Fig. 7.3 with other quantities to be discussed later.

A study of the plot indicates that the failure rate (number of failures per hour) is, in fact, decreasing and hence NHPP (Model FC1) can be employed to model the failure process.

Step 2.

Using the method of maximum likelihood (See Appendix D-1), estimates of the parameter a and b were obtained and are

Table 7-1
 FAILURES IN ONE HOUR (EXECUTION TIME)
 INTERVALS AND CUMULATIVE FAILURES

Hour	SYS1	
	No.	Cum.
1	27	27
2	16	43
3	11	54
4	10	64
5	11	75
6	7	82
7	2	84
8	5	89
9	3	92
10	1	93
11	4	97
12	7	104
13	2	106
14	5	111
15	5	116
16	6	122
17	0	122
18	5	127
19	1	128
20	1	129
21	2	131
22	1	132
23	2	134
24	1	135
25	1	136

AD-A139 240

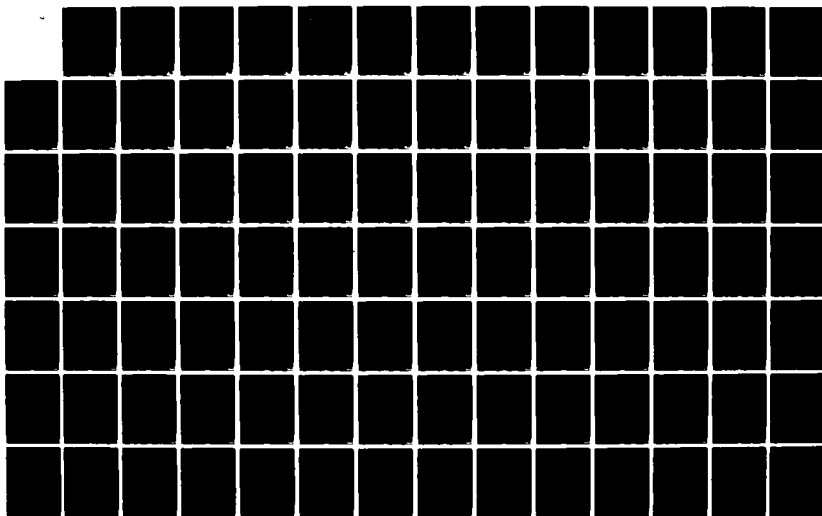
A GUIDEBOOK FOR SOFTWARE RELIABILITY ASSESSMENT(U)
SYRACUSE UNIV NY A L GOEL AUG 83 RADC-TR-83-176
F30602-81-C-0169

23

UNCLASSIFIED

F/G 9/2

NL





—

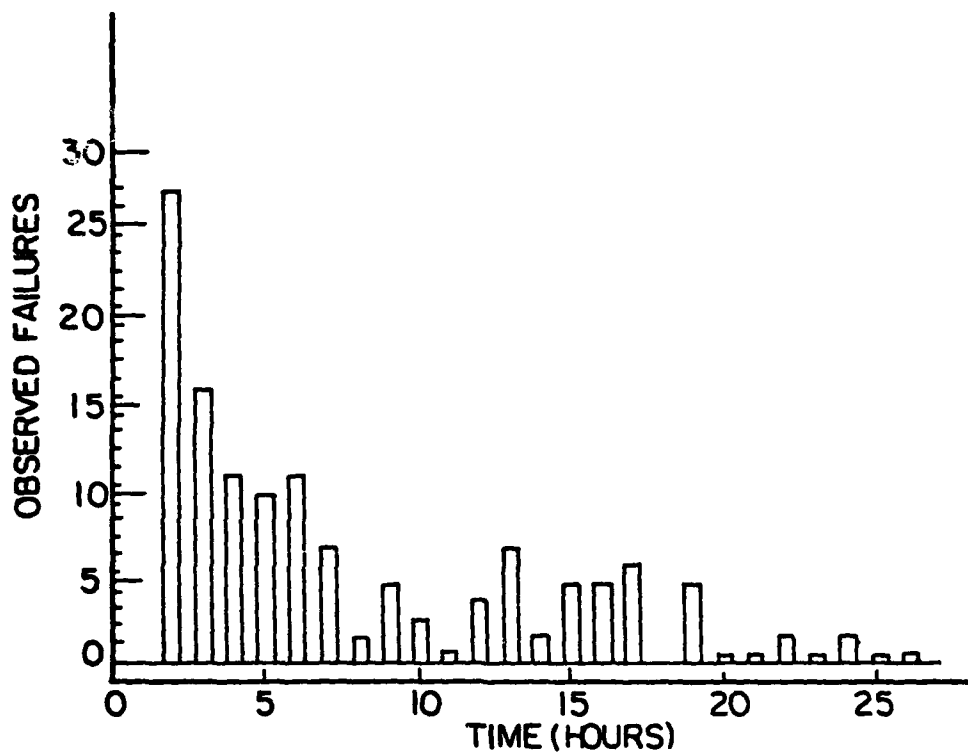


Fig. 7.2. Plot of the Number of Failures Per Hour (SYS1)

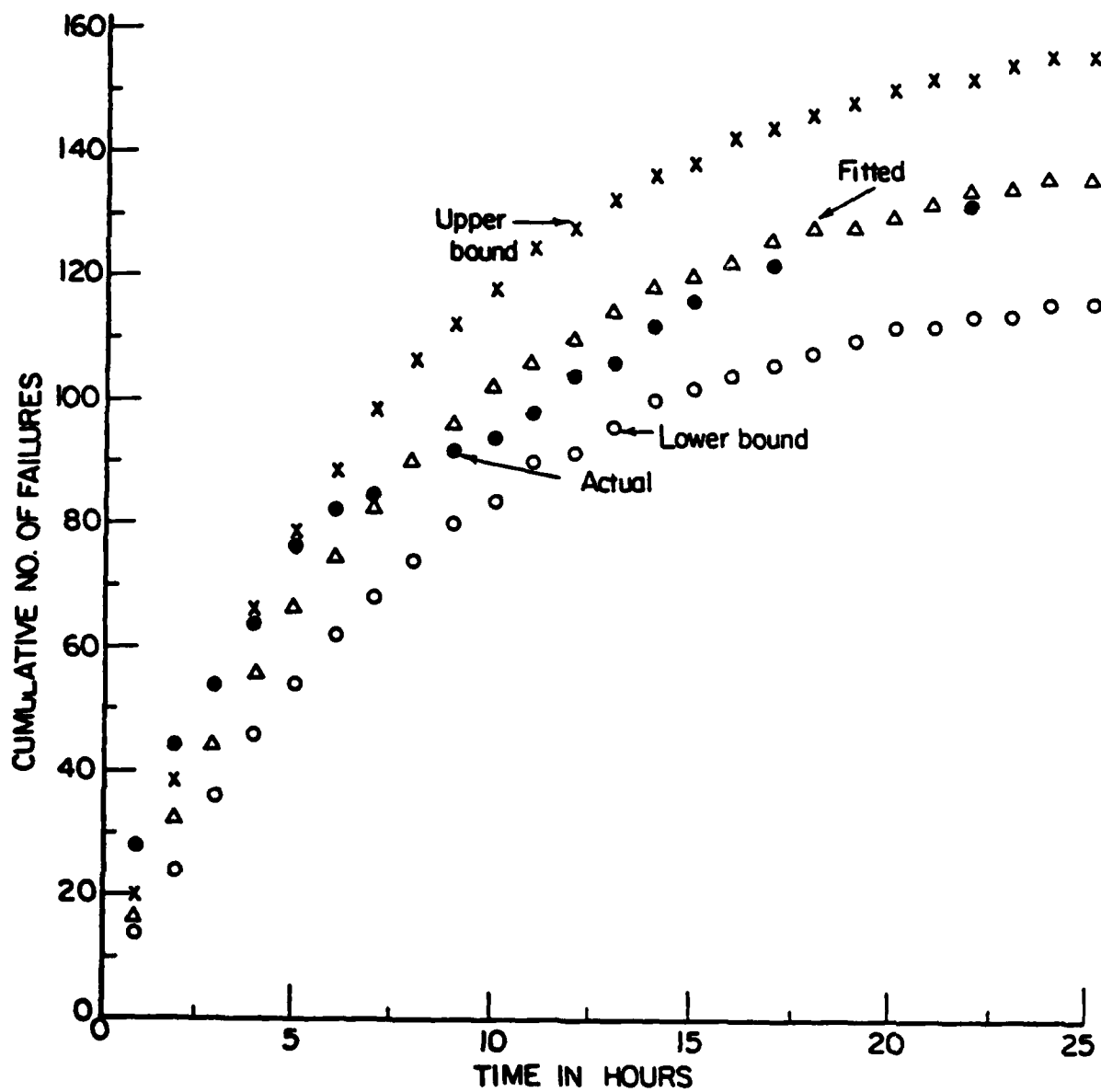


Fig. 7.3. Number of Failures and 90% Confidence Bounds (SYS1)

$$\hat{a} = 142.32 \text{ (Expected number of faults)}$$

$$\hat{b} = 0.1246 \text{ (Faults per fault per hour)}$$

For details of these computations, see Section 7.3.

Step 3

The model based on the data of Table 7.1 is

$$\hat{m}(t) = 142.32(1 - e^{-0.1246t})$$

and

$$\hat{\lambda}(t) = 17.73 \cdot e^{-0.1246t}$$

A plot of $\hat{m}(t)$ versus t is shown in Figure 7.3

Step 4.

We used the Kolmogorov-Smirnov goodness of fit test for checking the adequacy of the model. For details of this test see Goel [GOE82]. Details of the test for this data set are shown in Table 7.2. We note that on the basis of this test the model seems to indicate a good fit.

Step 5.

The confidence regions for \hat{a} and \hat{b} were computed but are not shown here.

Step 6.

We computed three performance measures: the cumulative number of failures, the expected number of remaining faults and software reliability. These measures are plotted in Figures 7.3, 7.4, and 7.5, respectively.

Plots of the confidence bounds for the expected number of failures, expected number of remaining faults and reliability are also shown in the above figures.

Table 7.2
Kolmogorov-Smirnoff Test for Data Set of Table 7.1

t_i	$H(t_i)$	$G_0(t_i)$	$ G_0(t_i) - H(t_i) $	$ G_0(t_i) - H(t_{i-1}) $
1	.198	.123	.076	.123
2	.316	.231	.085	.032
3	.397	.326	.071	.010
4	.470	.410	.060	.014
5	.551	.485	.066	.014
6	.602	.551	.051	.000
7	.617	.609	.008	.006
8	.654	.660	.006	.043
9	.676	.705	.029	.051
10	.683	.745	.061	.068
11	.713	.780	.067	.096
12	.764	.811	.047	.098
13	.779	.839	.059	.074
14	.816	.863	.047	.084
15	.852	.885	.032	.068
16	.897	.903	.006	.050
17	.897	.920	.023	.023
18	.933	.935	.001	.038
19	.941	.948	.007	.014
20	.948	.959	.011	.018
21	.963	.969	.006	.021
22	.970	.978	.008	.015
23	.985	.986	.001	.016
24	.992	.993	.001	.008
25	1	1	0	.007

$$\left. \begin{aligned} D_{25, .20} &= .208 > D_{\max} = 0.123 \\ D_{25, .05} &= .264 > D_{\max} = 0.123 \end{aligned} \right\}$$

Hence the model fits the data

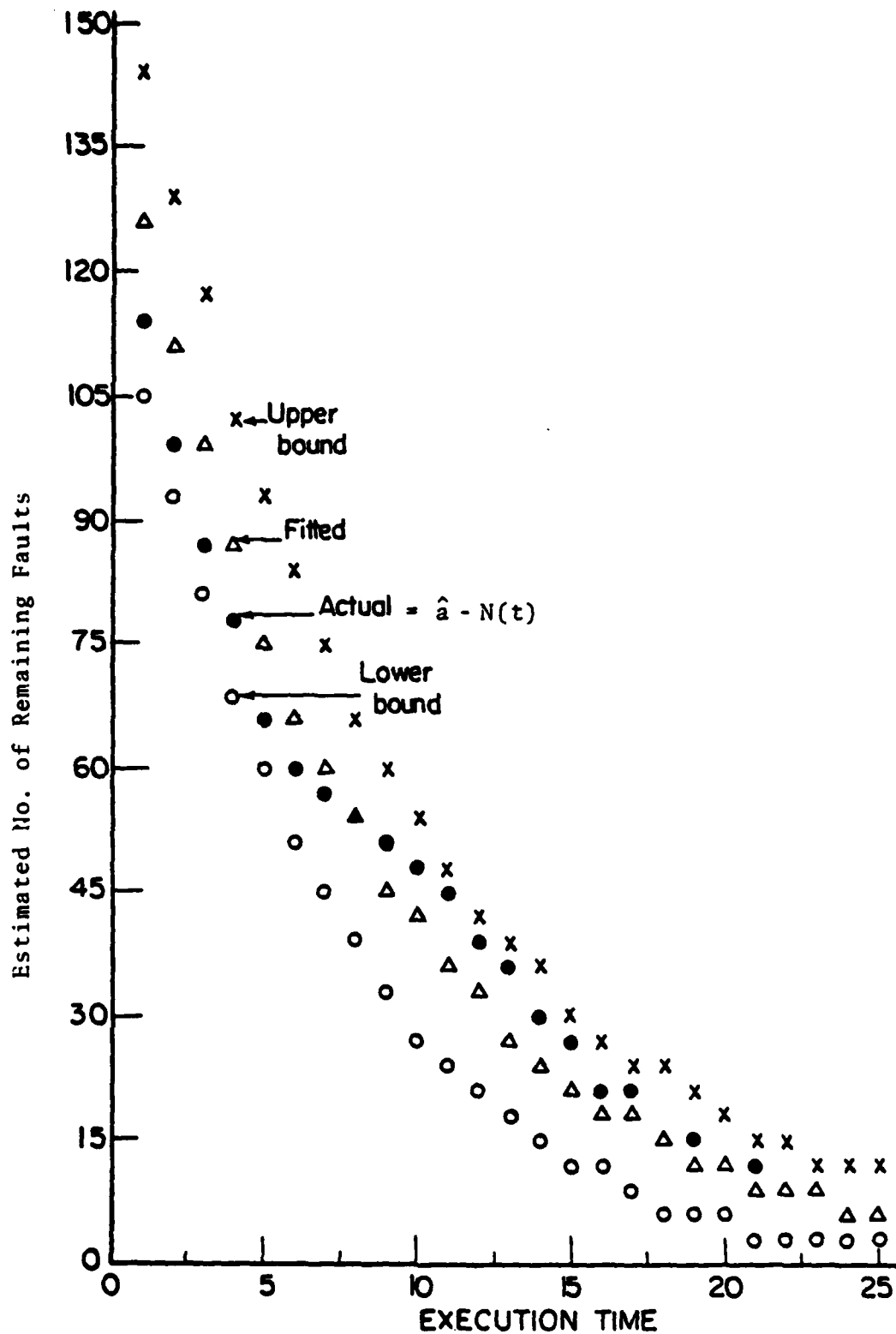


Fig. 7.4. Observed and Expected No. of Remaining Faults with 90% Confidence Bounds on $[E N(t)]$

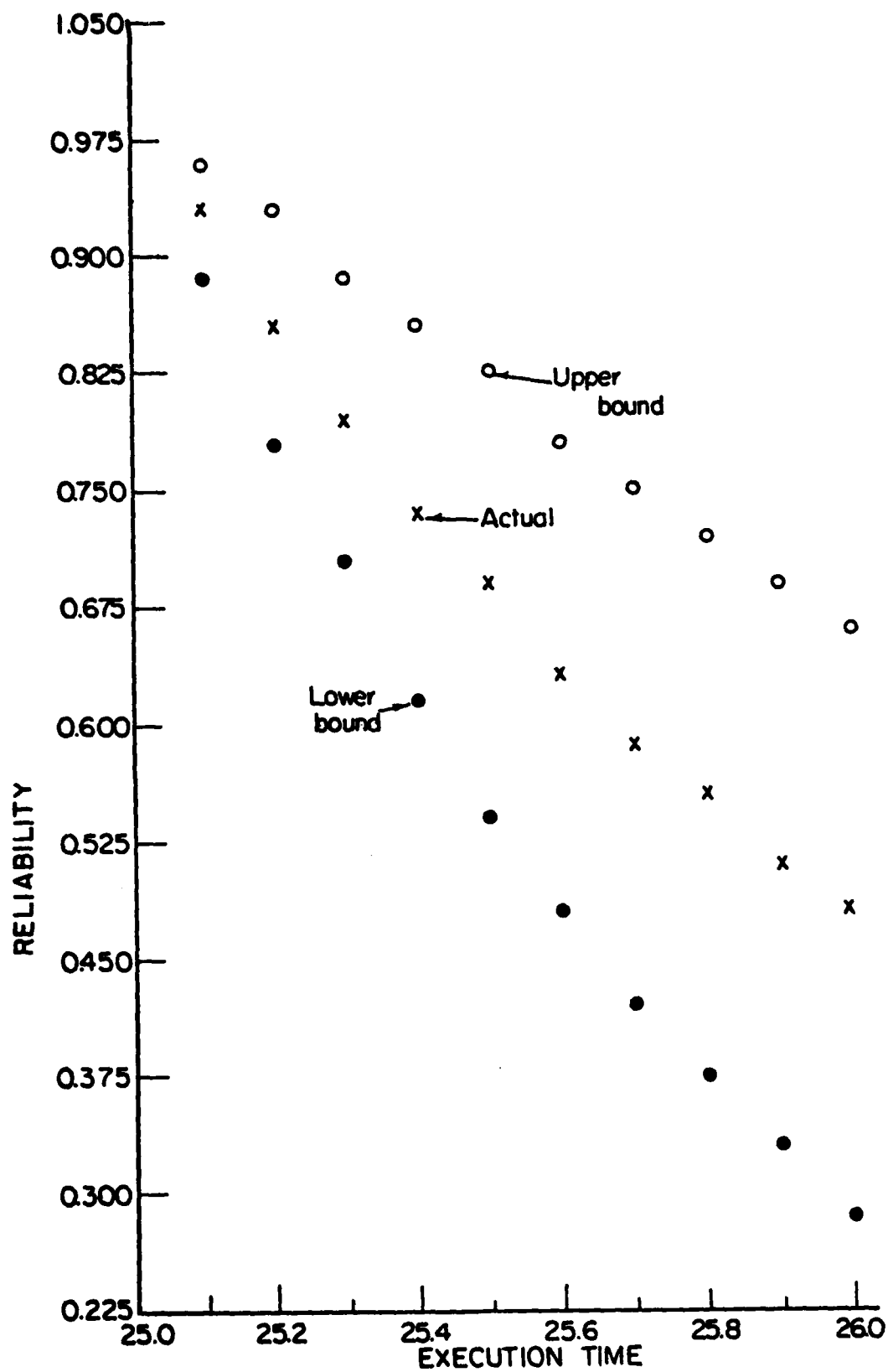


Fig. 7.5. Reliability and 90% Confidence Bounds - SYS1

A study of these plots indicates that the NHPP model (Model FCl) provides an excellent fit to the data and can be used for purposes of describing the failure behavior as well as prediction of future failure process. The information available from this can be used for planning, scheduling, and other management decisions.

Step 8

A plot of the reliability growth is shown in Figure 7.6. To obtain this plot, we recomputed the parameters a and b at one hour intervals based on data from 15, 16, ..., 25 hours of execution time testing. The one hour ahead predicted reliability at each of these points is what is shown in Figure 7.6.

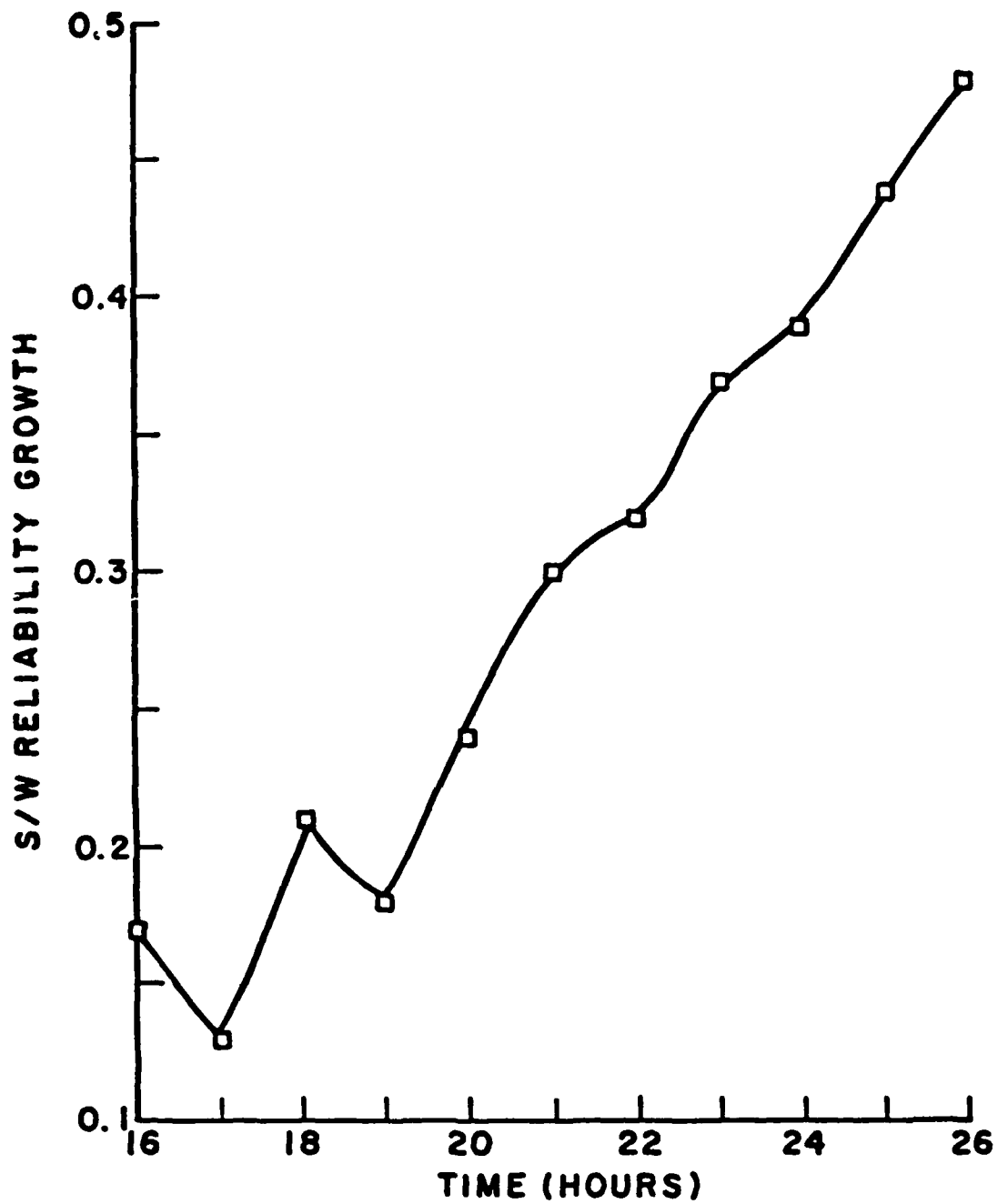


Fig. 7,6 S/W Reliability Growth with
Parameter Updating

7.3 Details of Parameter Estimation for the Data of Section 7.2.

The data in Section 7.2 can be written as follows,

Times t_i (x 3600 secs)	Cumulative # of Failures (y_i , $i = 1, 2, \dots, 25$)
1	27
2	43
3	54
4	64
.	
.	
.	
24	135
25	136

Estimation of Parameters a and b

MLE of a and b can be obtained by solving the following pair of equations

$$a(1 - e^{-bt_n}) = y_n$$

$$at_n e^{-bt_n} = \sum_{i=1}^n \frac{(y_i - y_{i-1})(t_i e^{-bt_i} - t_{i-1} e^{-bt_{i-1}})}{e^{-bt_{i-1}} - e^{-bt_i}}$$

The above two equations yield

$$\frac{y_n t_n e^{-bt_n}}{(1 - e^{-bt_n})} = \sum_{i=1}^n \frac{(y_i - y_{i-1})(t_i e^{-bt_i} - t_{i-1} e^{-bt_{i-1}})}{e^{-bt_{i-1}} - e^{-bt_i}}$$

MLE of b can be obtained from this using Newton-Raphson method.

Then we have

$$\hat{a} = \frac{y_n}{(1 - e^{-\hat{b}t_n})}$$

Use of the Newton-Raphson method to find \hat{b}

Let

$$F \equiv \sum_{i=1}^n \frac{(y_i - y_{i-1})(t_i e^{-bt_i} - t_{i-1} e^{-bt_{i-1}})}{e^{-bt_{i-1}} - e^{-bt_i}} - \frac{y_n t_n e^{-bt_n}}{1 - e^{-bt_n}}$$

then

$$\begin{aligned} \frac{dF}{db} \equiv \sum_{i=1}^n \frac{(y_i - y_{i-1})[(e^{-bt_{i-1}} - e^{-bt_i})(t_{i-1}^2 e^{-bt_{i-1}} - t_i^2 e^{-bt_i}) - (t_i e^{-bt_i} - t_{i-1} e^{-bt_{i-1}})^2]}{(e^{-bt_{i-1}} - e^{-bt_i})^2} \\ + \frac{y_n t_n^2 e^{-bt_n}}{(1 - e^{-bt_n})^2} \end{aligned}$$

For the above data set

$$n = 25, t_0 = 0, t_i = i \text{ for } i = 1, 2, \dots, 25$$

$$y_0 = 0, y_1 = 27, y_2 = 43, \dots, y_{24} = 135, y_{25} = 136.$$

Iteration 1

Let initial $b = b^{(1)} = 0.01$ then

$$\begin{aligned} F &= \sum_{i=1}^{25} \frac{(y_i - y_{i-1})(t_i e^{-.01t_i} - t_{i-1} e^{-.01t_{i-1}})}{e^{-.01t_{i-1}} - e^{-.01t_i}} - \frac{y_{25} t_{25} e^{-.25}}{(1 - e^{-0.25})} \\ &= 12665.11333 - \frac{136 \times 25 e^{-.25}}{(1 - e^{-.25})} \end{aligned}$$

$$= 12665.11333 - 11970.75966 = 694.35367$$

$$\begin{aligned} \frac{dF}{db} &= \sum_{i=1}^{25} \frac{(y_i - y_{i-1})[(e^{-.01t_{i-1}} - e^{-.01t_i})(t_{i-1}^2 e^{-.01t_{i-1}} - t_i^2 e^{-.01t_i}) - (t_i e^{-.01t_i} - t_{i-1} e^{-.01t_{i-1}})^2]}{(e^{-.01t_{i-1}} - e^{-.01t_i})^2} \\ &\quad + \frac{y_{25} t_{25}^2 e^{-.01t_{25}}}{(1 - e^{-.01t_{25}})^2} \end{aligned}$$

$$= -1359988.667 + 1352938.747$$

$$= -7049.920$$

$$\Delta b^{(1)} = -F \div \frac{dF}{db} = -694.35367 \div (-7049.920)$$

$$= 0.0984910087 > 0.0001 \Rightarrow \text{Go to Iteration 2}$$

Iteration 2

$$b^{(2)} = b^{(1)} + \Delta b^{(1)} = .01 + 0.0984910087$$

$$= .1084910087 \approx .108$$

$$F = \sum_{i=1}^{25} \frac{(y_i - y_{i-1})(t_i e^{-.108t_i} - t_{i-1} e^{-.108t_{i-1}})}{(e^{-.108t_{i-1}} - e^{-.108t_i})} - \frac{136 \times 25 e^{-.108 \times 25}}{1 - e^{-.108 \times 25}}$$

$$= 319.7894262 - 241.7604258 = 78.029000$$

$$\frac{dF}{db} = \sum_{i=1}^{25} \frac{(y_i - y_{i-1})[(e^{-.108t_{i-1}} - e^{-.108t_i})(t_{i-1}^2 e^{-.108t_{i-1}} - t_i^2 e^{-.108t_i}) - (t_i e^{-.108t_i} - t_{i-1} e^{-.108t_{i-1}})^2]}{(e^{-.108t_{i-1}} - e^{-.108t_i})^2} + \frac{136 \times 25^2 \times e^{-.108 \times 25}}{(1 - e^{-.108 \times 25})^2}$$

$$= -11543.18018 + 6473.77611$$

$$= -5069.40407$$

$$\Delta b^{(2)} = -F \div \frac{dF}{db} = -78.0290004 \div (-5069.40407)$$

$$= .0153921446 > .0001 =$$

$$\Rightarrow \text{Go to Iteration 3}$$

Iteration 3

$$b^{(3)} = b^{(2)} + \Delta b^{(2)} = .1084910087 + .0153921446$$

$$= .1238831533 \approx .124$$

$$F = \sum_{i=1}^{25} \frac{(y_i - y_{i-1})(t_i e^{-.124t_i} - t_{i-1} e^{-.124t_{i-1}})}{(e^{-.124t_{i-1}} - e^{-.124t_i})} - \frac{136 \times 25 e^{-.124 \times 25}}{(1 - e^{-.124 \times 25})}$$

$$= 164.2123219 - 160.8842808 = 3.3280411$$

$$\frac{dF}{db} = \sum_{i=1}^{25} \frac{(y_i - y_{i-1}) (e^{-.124t_{i-1}} - e^{-.124t_i}) (t_i^2 e^{-.124t_{i-1}} - t_{i-1}^2 e^{-.124t_i})}{-(t_i e^{-.124t_i} - t_{i-1} e^{-.124t_{i-1}})^2} + \frac{136 \times 25^2 \times e^{-.124 \times 25}}{(1 - e^{-.124 \times 25})^2}$$

$$= -8850.321535 + 4212.428725$$

$$= -4637.892810$$

$$\Delta b^{(3)} = -F \div \frac{dF}{db} = -3.3280411 \div (-4637.892810)$$

$$= 7.175761232 \times 10^{-4} > 0.0001 \Rightarrow \text{Go to Iteration 4}$$

Iteration 4

$$b^{(4)} = b^{(3)} + \Delta b^{(3)} = .1238831533 + .0007175761232$$

$$= .1246007294 \approx .125$$

$$F = \sum_{i=1}^{25} \frac{(y_i - y_{i-1}) (t_i e^{-.125t_i} - t_{i-1} e^{-.125t_{i-1}})}{(e^{-.125t_{i-1}} - e^{-.125t_i})^2} - \frac{136 \times 25 \times e^{-.125 \times 25}}{(1 - e^{-.125 \times 25})}$$

$$= 157.8981635 - 157.8910124 = 0.0071511$$

$$\frac{dF}{db} = \sum_{i=1}^{25} \frac{(y_i - y_{i-1}) (e^{-.125t_{i-1}} - e^{-.125t_i}) (t_i^2 e^{-.125t_{i-1}} - t_{i-1}^2 e^{-.125t_i})}{-(t_i e^{-.125t_i} - t_{i-1} e^{-.125t_{i-1}})^2} + \frac{136 \times 25^2 \times e^{-.125 \times 25}}{(1 - e^{-.125 \times 25})^2}$$

$$= -8748.547027 + 4130.580986$$

$$= -4617.966041$$

$$\Delta b^{(4)} = -F \div \frac{dF}{db} = -.0071511 \div (-46.7.966041)$$

$$= 1.5485388 \times 10^{-6} < 0.0001 \Rightarrow$$

$$\hat{b} = b^{(4)} + \Delta b^{(4)} = .1246007294 + .0000015485$$

$$= .1246022779 \approx .1246$$

$$b) \hat{a} = \frac{y_{25}}{(1 - e^{-\hat{b} \times 25})} = \frac{136}{(1 - e^{-\hat{b} \times 25})} = \frac{136}{(1 - e^{-.1246 \times 25})} = 142.3156405$$

7.4 Numerical Examples

Once a model has been identified, the key step in fitting the model to the observed failure data is the estimation of parameters. This requires detailed computations because in almost all cases the likelihood equations have to be solved iteratively using numerical techniques.

In this section we show the details of such computations for selected data sets. Two types of data are considered viz, times between failures and failure counts. Computations for some performance measures are also given. The two models considered are the Jelinski-Moranda De-eutrophication Model (Model TBF1) and the Goel-Okumoto NHPP model (Model FC1).

The data set consists of only a few points so that the computations could be repeated using a hand calculator. It should be pointed out that the leading decimal places have been retained in these examples to ensure that the round-off errors do not build up.

7.4.1 Example showing computations for the Jelinski-Moranda Model (Model TBF1)

Consider a set of failure times as follows:

<u>Failure no.</u>	<u>Time Between Failures t_i</u>
1	3
2	30
3	113
4	31
5	115

The parameters to be estimated are N and ϕ

The maximum likelihood estimate of N can be obtained from the following equation using Newton-Raphson method:

$$\sum_{i=1}^n \frac{1}{N - (i-1)} - \frac{n}{N - \frac{1}{\sum_{i=1}^n t_i} \left(\sum_{i=1}^n (i-1)t_i \right)} = 0$$

We can then find $\hat{\phi}$ from the following equation by substituting \hat{N} for N

$$\hat{\phi} = \frac{n}{\hat{N} \left(\sum_{i=1}^n t_i \right) - \sum_{i=1}^n (i-1)t_i}$$

Use of Newton-Raphson method to find \hat{N}

Let

$$F = \sum_{i=1}^n \frac{1}{N - (i-1)} - \frac{n}{N - \frac{1}{\sum_{i=1}^n t_i} \left(\sum_{i=1}^n (i-1)t_i \right)}$$

then

$$\frac{dF}{dN} = \frac{n}{\left\{ N - \frac{1}{\sum_{i=1}^n t_i} \left(\sum_{i=1}^n (i-1)t_i \right) \right\}^2} - \sum_{i=1}^n \frac{1}{\{N - (i-1)\}^2}$$

For the above dataset $n = 5$, $\sum_{i=1}^n t_i = \sum_{i=1}^5 t_i =$
 $= (3 + 30 + 113 + 31 + 115) = 292$; and

$$\sum_{i=1}^n (i-1)t_i = \sum_{i=1}^5 (i-1)t_i = t_2 + 2t_3 + 3t_4 + 4t_5$$

$$= 30 + 226 + 93 + 460 = 809$$

Iteration 1: Let an initial value of N be $N^{(1)} = 5$

$$\text{Then } F^{(1)} = \sum_{i=1}^5 \frac{1}{5 - i + 1} - \frac{5}{5 - \frac{1}{292} \cdot (809)}$$

$$= \left(\frac{1}{5} + \frac{1}{4} + \frac{1}{3} + \frac{1}{2} + 1\right) - \frac{5}{2.229452055}$$

$$= 2.283333333 - 2.242703532$$

$$= 4.062980 \times 10^{-2}$$

and

$$\frac{dF^{(1)}}{dN^{(1)}} = \frac{5}{\left\{5 - \frac{1}{292} (809)\right\}^2} - \sum_{i=1}^5 \frac{1}{\{5 - i + 1\}^2}$$

$$= \frac{5}{\{2.229452055\}^2} - \left(\frac{1}{25} + \frac{1}{16} + \frac{1}{9} + \frac{1}{4} + 1\right)$$

$$\text{or } \frac{dF^{(1)}}{dN^{(1)}} = -4.57667284 \times 10^{-1}$$

Next

$$\Delta N^{(1)} = - F^{(1)} \div \frac{dF^{(1)}}{dN^{(1)}} = (4.062980 \times 10^{-2}) \div (4.57667284 \times 10^{-1})$$

$$= 0.08877584602$$

Let error $\leq 10^{-4}$. Since $\Delta N^{(1)} \not\leq 10^{-4}$, we go through the next iteration.

Iteration 2. The value of N for the second iteration is

$$N^{(2)} = N^{(1)} + \Delta N^{(1)} = 5.08877584602$$

$$F^{(2)} = \sum_{i=1}^5 \frac{1}{5.08877584602 - i + 1} - \frac{5}{5.08877584602 - \frac{809}{292}}$$

$$\text{or } F^{(2)} = 5.22788908141 \times 10^{-3}$$

and

$$\frac{dF^{(2)}}{dN^{(2)}} = \frac{5}{\left\{5.08877584602 - \frac{809}{292}\right\}^2} - \sum_{i=1}^5 \frac{1}{\left\{5.08877584602 - i + 1\right\}^2}$$

$$= .9303743904 - 1.276022534 = -3.45648143667 \times 10^{-1}$$

Now

$$\begin{aligned} \Delta N^{(2)} &= -F^{(2)} \div \frac{dF^{(2)}}{dN^{(2)}} = (5.22788908141 \times 10^{-3}) \div (3.45648143667 \times 10^{-1}) \\ &= 0.0151248869 \end{aligned}$$

Since this number is $>10^{-4}$, we go to the next iteration.

Iteration 3. Value of N for the third iteration is

$$\begin{aligned} N^{(3)} &= N^{(2)} + \Delta N^{(2)} = 5.08877584602 + 0.0151248869 \\ &= 5.10390073293 \end{aligned}$$

$$\begin{aligned} F^{(3)} &= \sum_{i=1}^5 \frac{1}{5.10390073293 - i + 1} - \frac{5}{5.10390073293 - \frac{809}{292}} \\ &= 2.142960589 - 2.142839277 = 1.21312176929 \times 10^{-4} \end{aligned}$$

and

$$\begin{aligned} \frac{dF^{(3)}}{dN^{(3)}} &= \frac{5}{\left\{5.10390073293 - \frac{809}{292}\right\}^2} - \sum_{i=1}^5 \frac{1}{\left\{5.10390073293 - i + 1\right\}^2} \\ &= .9183520332 - 1.248093532 = -3.29741499203 \times 10^{-1} \end{aligned}$$

$$\begin{aligned} \Delta N^{(3)} &= -F^{(3)} \div \frac{dF^{(3)}}{dN^{(3)}} = (1.21312176929 \times 10^{-4}) \div (3.29741499203 \times 10^{-1}) \\ &= 3.679008472 \times 10^{-4} \end{aligned}$$

Since $\Delta N^{(3)} > 10^{-4}$, we continue the computations

Iteration 4. The value of N for the iteration

$$\begin{aligned} N^{(4)} &= N^{(3)} + \Delta N^{(3)} = 5.10390073293 + 3.679008472 \times 10^{-4} \\ &= 5.10426863377 \end{aligned}$$

$$\begin{aligned} F^{(4)} &= \sum_{i=1}^5 \frac{1}{5.10426863377 - i + 1} - \frac{5}{5.10426863377 - \frac{809}{292}} \\ &= 2.142501537 - 2.142501468 = 6.93541917229 \times 10^{-8} \end{aligned}$$

$$\begin{aligned} \frac{dF^{(4)}}{dN^{(4)}} &= \frac{5}{\{5.10426863377 - \frac{809}{292}\}^2} - \sum_{i=1}^5 \frac{1}{\{5.10426863377 - i + 1\}^2} \\ &= 0.9180625077 - 1.247427058 = -3.29364549966 \times 10^{-1} \end{aligned}$$

and

$$\begin{aligned} \Delta N^{(4)} &= -F^{(4)} \div \frac{dF^{(4)}}{dN^{(4)}} = (6.9354191722 \times 10^{-8}) \div (3.29364549966 \times 10^{-1}) \\ &= 2.105696916 \times 10^{-7} \end{aligned}$$

Since $\Delta N^{(4)} < 10^{-4}$ we terminate further iterations and compute \hat{N} as

$$\begin{aligned} \hat{N} &= N^{(4)} + \Delta N^{(4)} \\ &= 5.10426863377 + 2.105696916 \times 10^{-7} \end{aligned}$$

or $\hat{N} = \underline{5.104268844}$

Using the above value of \hat{N} , we get

$$\hat{\phi} = \frac{5}{5.104268844 \cdot (292) - 809}$$

$$= \frac{5}{1490.446502 - 809} = 5/681.446502$$

or $\hat{\phi} = \underline{7.337333131 \times 10^{-3}}$

For this data set, we get

$$\hat{N} = 5.104268844$$

$$\hat{\phi} = 7.337333131 \times 10^{-3}$$

Performance Measures

After estimating the parameters N and ϕ , we calculate the performance measures using these estimates as follows:

- Reliability at time t after n th failure $\hat{R}_n(t)$

$\hat{R}_n(t)$ is obtained using the following formula

$$\hat{R}_n(t) = \exp[-\hat{\phi} (\hat{N}-n)t]$$

Let $t=10$. Then substituting values for $\hat{N}, \hat{\phi}, n$, and t in the above formula we get

$$\hat{R}_5(10) = \exp[-7.337333131 \times 10^{-3} \times (5.104268844-5) \times 10]$$

$$= .9923786386$$

- Mean Time to Failure After n th Failure \hat{MTTF}_n

\hat{MTTF}_n is obtained using the following formula:

$$\hat{MTTF}_n = \frac{1}{\hat{\phi} (\hat{N}-n)}$$

Substituting values for $\hat{N}, \hat{\phi}$ and n in the above formula we get

$$\hat{MTTF}_5 = \frac{1}{7.337333131 \times 10^{-3} \times (5.104268844-5)}$$

$$= 1307.095152$$

7.4.2 Examples showing computations for the Goel-Okumoto Model (Model FC1)

Consider the following failure counts based on the data of the previous example.

<u>Interval No.</u>	<u>Cumulative No. of failures</u>
1	2
2	4
3	5

The parameters to be estimated are a and b.

MLE of a and b can be obtained by solving the following pair of equations

$$\begin{aligned} a(1-e^{-bt_n}) &= y_n \\ at_n e^{-bt_n} &= \sum_{i=1}^n \frac{(y_i - y_{i-1})(t_i e^{-bt_i - t_{i-1}} e^{-bt_{i-1}})}{e^{-bt_{i-1}} - e^{-bt_i}} \end{aligned}$$

These equations yield

$$\frac{y_n t_n e^{-bt_n}}{(1-e^{-bt_n})} = \sum_{i=1}^n \frac{(y_i - y_{i-1})(t_i e^{-bt_i - t_{i-1}} e^{-bt_{i-1}})}{e^{-bt_{i-1}} - e^{-bt_i}}$$

which can be used to obtain the mle of b by the Newton-Raphson method. Then we can compute the mle of a as

$$\hat{a} = \frac{y_n}{(1-e^{-b\hat{t}_n})}$$

Use of Newton-Raphson method to find \hat{b}

$$\text{Let } F \equiv \sum_{i=1}^n \frac{(y_i - y_{i-1})(t_i e^{-bt_i - t_{i-1}} e^{-bt_{i-1}})}{e^{-bt_{i-1}} - e^{-bt_i}} - \frac{y_n t_n e^{-bt_n}}{(1-e^{-bt_n})}$$

then

$$\begin{aligned} \frac{dF}{db} &= \sum_{i=1}^n \frac{(y_i - y_{i-1})[(e^{-bt_{i-1}} - e^{-bt_i})(t_{i-1}^2 e^{-bt_{i-1}} - t_i^2 e^{-bt_i})]}{(e^{-bt_{i-1}} - e^{-bt_i})^2} \\ &\quad - \frac{(t_i e^{-bt_i} - t_{i-1} e^{-bt_{i-1}})^2}{(e^{-bt_{i-1}} - e^{-bt_i})^2} + \frac{y_n t_n^2 e^{-bt_n}}{(1-e^{-bt_n})^2} \end{aligned}$$

For the above data set

$$n = 3, t_0 = 0, t_1 = 1, t_2 = 2, t_3 = 3$$

$$y_0 = 0, y_1 = 2, y_2 = 4, y_3 = 5$$

Iteration 1. Let an initial value of $b = .01$, i.e., $b^{(1)} = 0.01$

Then

$$\begin{aligned} F &= \sum_{i=1}^3 \frac{(y_i - y_{i-1})(t_i e^{-.01t_i} - t_{i-1} e^{-.01t_{i-1}})}{e^{-.01t_{i-1}} - e^{-.01t_i}} - \frac{y_3 t_3 e^{-.01t_3}}{(1 - e^{-.01t_3})} \\ &= \frac{y_1(t_1 e^{-.01t_1})}{1 - e^{-.01t_1}} + \frac{(y_2 - y_1)(t_2 e^{-.01t_2} - t_1 e^{-.01t_1})}{e^{-.01t_1} - e^{-.01t_2}} \\ &\quad + \frac{(y_3 - y_2)(t_3 e^{-.01t_3} - t_2 e^{-.01t_2})}{e^{-.01t_2} - e^{-.01t_3}} - \frac{y_3 t_3 e^{-.01t_3}}{(1 - e^{-.01t_3})} \\ &= \frac{2(e^{-.01})}{1 - e^{-.01}} + \frac{(4 - 2)(2e^{-.02} - e^{-.01})}{e^{-.01} - e^{-.02}} + \frac{(5 - 4)(3e^{-.03} - 2e^{-.02})}{e^{-.02} - e^{-.03}} \\ &\quad - \frac{5 \cdot 3 \cdot e^{-.03}}{(1 - e^{-.03})} \end{aligned}$$

$$= 493.5041667 - 492.5374994$$

$$= .966667222210$$

Now,

$$\begin{aligned} \frac{dF}{db} &= \sum_{i=1}^3 \frac{(y_i - y_{i-1})[(e^{-.01t_{i-1}} - e^{-.01t_i})(t_{i-1}^2 e^{-.01t_{i-1}} - t_i^2 e^{-.01t_i}) - (t_i e^{-.01t_i} - t_{i-1} e^{-.01t_{i-1}})^2]}{(e^{-.01t_{i-1}} - e^{-.01t_i})^2} + \frac{y_3 t_3^2 e^{-.01t_3}}{(1 - e^{-.01t_3})^2} \\ &= \frac{y_1[(e^{-.01 \times 0} - e^{-.01})(-e^{-.01}) - e^{-.01 \times 2}]}{(e^{-.01 \times 0} - e^{-.01})^2} \end{aligned}$$

$$\begin{aligned}
& + \frac{(y_3 - y_2) [(e^{-.02} - e^{-.03}) (4e^{-.02} - 9e^{-.03}) - (3e^{-.03} - 2e^{-.02})^2]}{(e^{-.02} - e^{-.03})^2} \\
& + \frac{5.9 \cdot e^{-.03}}{(1 - e^{-.03})^2} \\
& + \frac{(y_2 - y_1) [(e^{-.01} - e^{-.02}) (e^{-.01} - 4e^{-.02}) - (2e^{-.02} - e^{-.01})^2]}{(e^{-.01} - e^{-.02})^2} \\
& = \frac{2[(1 - e^{-.01}) (-e^{-.01}) - e^{-.02}]}{(1 - e^{-.01})^2} \\
& + \frac{2[(e^{-.01} - e^{-.02}) (e^{-.01} - 4e^{-.02}) - (2e^{-.02} - e^{-.01})^2]}{(e^{-.01} - e^{-.02})^2} \\
& + \frac{1[(e^{-.02} - e^{-.03}) (4e^{-.02} - 9e^{-.03}) - (3e^{-.03} - 2e^{-.02})^2]}{(e^{-.02} - e^{-.03})^2} \\
& + \frac{45 \cdot e^{-.03}}{(1 - e^{-.03})^2} \\
& = - \left(\frac{2e^{-.01}}{(1 - e^{-.01})^2} + \frac{2e^{-.03}}{(e^{-.01} - e^{-.02})^2} + \frac{e^{-.05}}{(e^{-.02} - e^{-.03})^2} \right) + \frac{45e^{-.03}}{(1 - e^{-.03})^2} \\
& = -49999.58334 + 49996.25017 \\
& = -3.33316667270
\end{aligned}$$

and

$$\begin{aligned}
\Delta b^{(1)} & = -F \div dF/db = -.966667222210 \div -3.33316667270 \\
& = .2900146669 > .0001
\end{aligned}$$

Let $\epsilon = 0.0001$. Since $\Delta b^{(1)} > \epsilon$, we go to iteration 2.

Iteration 2. $b^{(2)} = b^{(1)} + \Delta b^{(1)} = .3000146669 \approx .3$

$$\begin{aligned}
F & = \sum_{i=1}^3 \frac{(y_i - y_{i-1}) (t_i e^{-.3t_i} - t_{i-1} e^{-.3t_{i-1}})}{e^{-.3t_{i-1}} - e^{-.3t_i}} - \frac{y_3 t_3 e^{-.3t_3}}{(1 - e^{-.3t_3})} \\
& = \frac{2(e^{-.3})}{1 - e^{-.3}} + \frac{2(2e^{-.6} - e^{-.3})}{e^{-.3} - e^{-.6}} + \frac{1(3e^{-.9} - 2e^{-.6})}{(e^{-.6} - e^{-.9})} - \frac{5.3e^{-.9}}{(1 - e^{-.9})}
\end{aligned}$$

$$= 10.2906087 - 10.27600431$$

$$= .0146665525129$$

Now,

$$\begin{aligned} \frac{dF}{db} &= \sum_{i=1}^3 \frac{(y_i - y_{i-1}) [(e^{-.3t_{i-1}} - e^{-.3t_i}) (t_{i-1}^2 e^{-.3t_{i-1}} - t_i^2 e^{-.3t_i})]}{(t_i e^{-.3t_i} - t_{i-1} e^{-.3t_{i-1}})^2} + \frac{y_3 t_3^2 e^{-.3t_3}}{(1 - e^{-.3t_3})^2} \\ &= \frac{2[(1 - e^{-.3})(-e^{-.3}) - e^{-.6}]}{(1 - e^{-.3})^2} \\ &\quad + \frac{2[(e^{-.3} - e^{-.6})(e^{-.3} - 4e^{-.6}) - (2e^{-.6} - e^{-.3})^2]}{(e^{-.3} - e^{-.6})^2} \\ &\quad + \frac{1[(e^{-.6} - e^{-.9})(4e^{-.6} - 9e^{-.9}) - (3e^{-.9} - 2e^{-.6})^2]}{(e^{-.6} - e^{-.9})^2} + \frac{45e^{-.9}}{(1 - e^{-.9})^2} \\ &= - \left(\frac{2e^{-.3}}{(1 - e^{-.3})^2} + \frac{2e^{-.9}}{(e^{-.3} - e^{-.6})^2} + \frac{e^{-1.5}}{(e^{-.6} - e^{-.9})^2} \right) + \frac{45e^{-.9}}{(1 - e^{-.9})^2} \\ &= -55.13532562 + 51.94726586 = -3.18805975584 \end{aligned}$$

$$\begin{aligned} \Delta b^{(2)} &= -F \div dF/db = -.0146665525129 \div -3.18805975584 \\ &= .004600463491 \end{aligned}$$

Since $\Delta b^{(2)} > \epsilon$, we continue with iteration 3.

$$\begin{aligned} \text{Iteration 3. } b^{(3)} &= b^{(2)} + \Delta b^{(2)} = .3000146669 + .004600463491 \\ &= .30461513036 \approx .305 \end{aligned}$$

$$\begin{aligned} F &= \sum_{i=1}^3 \frac{(y_i - y_{i-1}) (t_i e^{-.305t_i} - t_{i-1} e^{-.305t_{i-1}})}{e^{-.305t_{i-1}} - e^{-.305t_i}} - \frac{y_3 t_3^2 e^{-.305t_3}}{(1 - e^{-.305t_3})} \\ &= \frac{2(e^{-.305})}{1 - e^{-.305}} + \frac{2(2e^{-.610} - e^{-.305})}{e^{-.305} - e^{-.610}} + \frac{1(3e^{-.915} - 2e^{-.610})}{e^{-.610} - e^{-.915}} \\ &\quad - \frac{5.3 \cdot e^{-.915}}{(1 - e^{-.915})} \\ &= 10.04088223 - 10.04087226 = 9.96719302759 \times 10^{-6} \end{aligned}$$

$$\frac{dF}{db} = \sum_{i=1}^3 \frac{(y_i - y_{i-1}) [(e^{-.305t_{i-1}} - e^{-.305t_i}) (t_{i-1}^2 e^{-.305t_{i-1}} - t_i^2 e^{-.305t_i}) - (t_i e^{-.305t_i} - t_{i-1} e^{-.305t_{i-1}})^2]}{(e^{-.305t_{i-1}} - e^{-.305t_i})^2} + \frac{y_3 t_3^2 e^{-.305t_3}}{(1 - e^{-.305t_3})^2}$$

$$\begin{aligned} &= \frac{2[(1 - e^{-.305})(-e^{-.305}) - e^{-.610}]}{(1 - e^{-.305})^2} + \\ &+ \frac{2[(e^{-.305} - e^{-.610})(e^{-.305} - 4e^{-.610}) - (2e^{-.610} - e^{-.305})^2]}{(e^{-.305} - e^{-.610})^2} \\ &+ \frac{1[(e^{-.610} - e^{-.915})(4e^{-.610} - 9e^{-.915}) - (3e^{-.915} - 2e^{-.610})^2]}{(e^{-.610} - e^{-.915})^2} \\ &+ \frac{45 \cdot e^{-.915}}{(1 - e^{-.915})^2} \\ &= -\left(\frac{2e^{-.305}}{(1 - e^{-.305})^2} + \frac{2e^{-.915}}{(e^{-.305} - e^{-.610})^2} + \frac{e^{-1.525}}{(e^{-.610} - e^{-.915})^2}\right) \\ &\quad + \frac{45e^{-.915}}{(1 - e^{-.915})^2} \end{aligned}$$

$$= -53.470157 + 50.28643996 = -3.18371704092$$

$$\Delta b^{(3)} = -F \div dF/db = -9.96719302759 \times 10^{-6} \div -3.18371704092$$

$$= 3.1306781 \times 10^{-6}$$

Since $\Delta b^{(3)} < \epsilon$, we get

$$\hat{b} = b^{(3)} + \Delta b^{(3)} = 3.0461826104 \times 10^{-1}$$

and

b)

$$\hat{a} = \frac{y_3 - \hat{b}t_3}{(1 - e^{-\hat{b}t_3})} = \frac{5}{(1 - e^{-3.0461826104 \times 3})} = 8.346957421$$

For this data set

$$\hat{a} = 8.34695742$$

$$\hat{b} = 3.0461826104 \times 10^{-1}$$

Performance Measures

After estimation of the parameters a and b the next step is to obtain performance measures as follows:

- Expected Number of Faults Detected by Time t $E[\hat{N}(t)]$

$E[\hat{N}(t)]$ is obtained using the following formula:

$$E[\hat{N}(t)] = \hat{a}(1 - e^{-\hat{b}t})$$

Let $t=10$ then substituting values for \hat{a}, \hat{b} , and t in the above formula we get

$$\begin{aligned} E[\hat{N}(10)] &= 8.34695742 \times (1 - e^{-3.0461826104 \times 10^{-1} \times 10}) \\ &= 7.950397851. \end{aligned}$$

- Expected Number of Remaining Faults by Time t , $E[\hat{\bar{N}}(t)]$

$E[\hat{\bar{N}}(t)]$ is obtained using the following formula

$$E[\hat{\bar{N}}(t)] = \hat{a}e^{-\hat{b}t}.$$

Let $t=10$ then substituting values for \hat{a}, \hat{b} , and t in the above formula we get

$$\begin{aligned} E[\hat{\bar{N}}(10)] &= 8.3469742 \times (e^{-3.0461826104 \times 10^{-1} \times 10}) \\ &= 8.34695742 \times e^{-3.0461826104} \\ &= 0.3965595689 \end{aligned}$$

- Reliability at Time t after n th testing Interval $\hat{R}_n(t)$

$\hat{R}_n(t)$ is obtained from the following formula

$$\hat{R}_n(t) = \exp[-\hat{a}\{\exp[-\hat{b}xn] - \exp[-\hat{b}x(n+t)]\}]$$

For the above example $n=3$. Let t be 1. After substituting values of \hat{a} , \hat{b} , n , and t in the above formula we get

$$\begin{aligned} \hat{R}_3(1) &= \exp[-8.34695742 \times \{\exp[-0.30461826104 \times 3] \\ &\quad - \exp[-0.30461826104 \times (3+1)]\}] = 0.4152494843 \end{aligned}$$

- Mean Time to Failure After nth Testing interval \hat{MTTF}_n

\hat{MTTF}_n is obtained using the following formula

$$\hat{MTTF}_n = \frac{1}{\hat{\lambda}(n)} = \frac{1}{\hat{a}\hat{b}\exp[-\hat{b}xn]}$$

For the above example $n=3$ and substituting values for \hat{a} and \hat{b} in the above formula we get

$$\begin{aligned}\hat{MTTF}_3 &= \frac{1}{8.34695742 \times 0.30461826104 \times \exp[-0.30461826104 \times 3]} \\ &= .9808216948\end{aligned}$$

8. CONCLUDING REMARKS

1. The objective of this report was to present information relevant to the selection and use of an analytical model for software reliability assessment. Towards this goal, we presented a summary of the available models in Sections 3, 4, and 5 which was backed up by detailed information in Appendices C, D, and E.
2. An important step in selecting a model is to develop a framework for describing the software development environment which can be mapped onto a set of assumptions that are consistent with those of the selected model or models. This is a difficult task and requires a clear understanding of the development environment as well as of the model assumptions. We feel that the material in Section 6 should be of great help in accomplishing this. It should, however, be pointed out that the assumptions required to formulate and develop an analytical model are rarely, if ever, satisfied in the real world. In other words, perfect adherence to model assumptions is an ideal which cannot be met in practice. A realistic approach, then, is to select a reliability model which captures the essence of the modelled environment.
3. It should also be pointed out that the arguments presented in Section 6.1 about the assumptions and in

Section 6.2 about the applicability of the models are based on our interpretation of the phases in the software development process. If particular situations differ from the scenarios presented there, appropriate modifications must be made in the model selection process.

9. References

- [ABR65] Abramowitz, M. and Stegun, I.A. (1965), Handbook of Mathematical Functions, Dover Publications, Inc.
- [ALL78] Allen, A.O. (1978), Probability, Statistics and Queueing Theory, Academic Press.
- [AND79] Anderson, T. and Randell, B. (Editors) (1979), Computing System Reliability, An Advanced Course, Cambridge University Press, London.
- [ANG80] Angus, J.E., Schafer, R.E., and Sukert, A. (1980), "Software Reliability Model Validation," Proc. Annual Reliability and Maintainability Symposium, San Francisco, CA, pp. 191-193.
- [AVI77] Avizienis, A. (1977), "Fault-Tolerant Computing: Progress, Problems, and Prospects," Proc. IFIP Congress 1977, Toronto, Canada, pp. 405-420.
- [BAR75] Barlow, R.E. and Proschan, F. (1975), Statistical Theory of Reliability and Life Testing: Probability Models, Holt, Rinehart and Winston, Inc.
- [BAS83] Basili, V.R., Selby, R.W. and Phillips, T. (1983), "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering (to appear).
- [BAS74] Basin, S.L. (1974), Estimation of Software Error Rate Via Capture-Recapture Sampling, Science Applications, Inc., Palo Alto, CA.
- [BAS80] Bastani, F.B. (1980), "An Input Domain Based Theory of Software Reliability and Its Application," Ph.D. Dissertation, University of California, Berkeley.
- [BEI79] Beightler, C.S., Phillips, D.T., and Wilde, D.J. (1979), Foundations of Optimization, Prentice-Hall.
- [BEL76] Belady, L.A. and Lehman, M.M. (1976), "A Model of Large Program Development," IBM Systems Journal, Vol. 15, No. 3, pp. 225-252.
- [BOE76] Boehm, B.W., Brown, J.R., and Lipow, M. (1976), "Quantitative Evaluation of Software Quality," Proc. 2nd International Conference on Software Engineering.
- [BRO75] Brown, J.R., Lipow, M. (1975), "Testing for Software Reliability," Proc. 1975 International Conference Reliable Software, Los Angeles, CA, pp. 518-527.

- [BRO80] Brooks, W.D. and Motley, R.W. (1980), "Analysis of Discrete Software Reliability Models," RADC-TR-80-84.
- [BRO72] Brown, M. (1972), "Statistical Analysis of Non-Homogeneous Poisson Processes," in Stochastic Point Processes, edited by P.A.W. Lewis, John Wiley & Sons, New York, pp. 67-89.
- [CAS80] Castillo, X. and Siewiorek, D.P. (1980), "A Performance Reliability Model for Computing Systems," Dept. of Computer Science, Carnegie-Mellon University.
- [CHA78] Champine, G.A. (1978), "What Makes a System Reliable," Datamation, pp. 195-206.
- [CHA78] Chandy, K.M. and Yeh, R.T. (Editors) (1978), Current Trends in Programming Methodology, Vol. III: Software Modeling, Prentice-Hall.
- [CHO80] Cho, C.K. (1980), An Introduction to Software Quality Control, John Wiley & Sons.
- [CIN75] Cinlar, E. (1975), Introduction to Stochastic Processes, Prentice-Hall.
- [CLA75] Clarke, L. (1975), "A System to Generate Test Data and Symbolically Execute Programs," Tech. Report #CU-CS-060-75, Dept. of Computer Science, Univ. of Colorado, Boulder.
- [COX65] Cox, D.R. and Miller, H.D. (1965), The Theory of Stochastic Processes, Wiley and Sons, Inc.
- [COX66] Cox, D.R. and Lewis, P.A.W. (1966), The Statistical Analysis of Series of Events, Methuen, London.
- [DAL77] Daly, E.B. (1977), "Management of Software Development," IEEE Transactions on Software Engineering, pp. 230-242.
- [DON75] Donelson, J., III, (1975), Duane's Reliability Growth Model as a Non-Homogeneous Poisson Process, IDA Log. No. HQ76-18012, Paper P-1162.
- [DUA64] Duane, J.T. (1964), "Learning Curve Approach to Reliability Monitoring," IEEE Transactions Aerospace, Vol. 2, pp. 563-566.

- [DUN82] Dunn, R. and Ullman, R. (1982), Quality Assurance for Computer Software, McGraw-Hill Book Company.
- [DUR80] Duran, J.W. and Wiorkowski, J.J. (1980), "Quantifying Software Validity by Sampling," IEEE Transactions on Reliability, Vol. R-29, No. 2.
- [END75] Endres, A. (1975), "An Analysis of Errors and Their Causes in System Programs," Proc. International Conference on Reliability Software, Los Angeles, CA, pp. 327-336.
- [FEL57] Feller, W. (1957), An Introduction to Probability Theory and Its Applications, 2nd Ed., Vol. I, John Wiley and Sons, Inc.
- [FEL66] Feller, W. (1966), An Introduction to Probability Theory and Its Applications, Vol. II, John Wiley and Sons, Inc.
- [FIN76] Finkelstein, J.M. (1976), "Confidence Bounds on the Parameters of the Weibull Process," Technometrics, Vol. 18, No. 1, pp. 115-117.
- [FOR77] Forman, E.H. and Singpurwalla, N.D. (1977), "An Empirical Stopping Rule for Debugging and Testing Computer Software," Journal of the American Statistical Association, Vol. 72, No. 360, pp. 750-757.
- [FRI77] Fries, M.J. (1977), Software Error Data Acquisition, Boeing Aerospace Co., Final Technical Report, RADCTR-77-130, AD A039-916.
- [GER76] Gerhart, S. and Yelowitz, L. (1976), "Observations of Fallibility in Applications of Modern Programming Methodologies," IEEE Transactions on Software Engineering.
- [GIL77] Gilb, T., Software Metrics, Winthrop Publishers, Inc., Massachusetts, 1977.
- [GIR73] Girard, E. and Rault, J.C. (1973), "A Programming Technique for Software Reliability," IEEE Symposium on Computer Software Reliability.
- [GLA79] Glass, R.L. (1979), Software Reliability Guidebook, Prentice-Hall, Inc.
- [GLA81] Glass, R.L. (1981), "Persistent Software Errors," IEEE Transactions on Software Engineering, Vol. SE-7.

- [GOE77] Goel, A.L. (1977), Summary of Technical Progress: Bayesian Software Reliability Prediction Models, RADC-TR-77-112, Syracuse University, AD A039-022.
- [GOE78a] Goel, A.L. and Okumoto, K. (1978), Bayesian Software Correction Limit Policies, Final Technical Report, Syracuse University, RADC-TR-78-155, Vol. 2 (of 5), AD A057-872.
- [GOE78b] Goel, A.L. and Okumoto, K. (1978), An Imperfect Debugging Model for Software Reliability, Final Technical Report, Syracuse University, RADC-TR-78-155, Vol. 1 (of 5), AD A057-879.
- [GOE78c] Goel, A.L., and Okumoto, K. (1978), "A Time Dependent Error Detection Rate Model for a Large Scale Software System," Proc. Third USA-Japan Computer Conference, San Francisco, CA, pp. 35-40.
- [GOE78d] Goel, A.L. and Okumoto, K. (1978), "An Analysis of Recurrent Software Failures in a Real-Time Control System," Proc. Annual Technical Conference, ACM, Washington, DC, pp. 496-500.
- [GOE79a] Goel, A.L. (1979), "Reliability and Other Performance Measures of Computer Software," Proc. First International Conference on Reliability and Exploitation of Computer Systems, Wroclaw, Poland, pp. 23-31.
- [GOE79b] Goel, A.L. and Okumoto, K. (1979), "A Time Dependent Error Detection Rate Model for Software Reliability and Other Performance Measures," IEEE Transactions on Reliability, Vol. R-28, No. 3, pp. 206-211.
- [GOE79c] Goel, A.L. and Okumoto, K. (1979), "A Markovian Model for Reliability and Other Performance Measures of Software Systems," Proc. National Computer Conference, New York, Vol. 48, pp. 769-774.
- [GOE80a] Goel, A.L. (1980), "A Software Error Detection Model with Applications," Journal of Systems and Software, Vol. 1, No. 3, pp. 243-249.
- [GOE80b] Goel, A.L. (1980), "A Summary of the Discussion on an Analysis of Computer Software Reliability Models," IEEE Transactions on Software Engineering, Vol. SE-6, No. 5, pp. 501-502.

- [GOE82a] Goel, A.L. (1982), "Software Reliability Modelling: An Overview and a Case Study," International Journal of Reliability and Safety.
- [GOE82b] Goel, A.L. (1982), Software Reliability Modelling and Estimation Techniques, RADC-TR-82-263.
- [GOE83] Goel, A.L., Basili, V.R., and Valdes, P.M. (1983), "When and How to Use a Software Reliability Model," Proc. Seventh Software Engineering Workshop, NASA/GSFC (to appear).
- [GOO77] Goodenough, J. and Gerhart, S. (1977), "Toward a Theory of Testing: Data Selection Criteria," Current Trends in Programming Methodology, Vol. 2, R.T. Yeh, Ed., Englewood Cliffs, NJ, Prentice-Hall.
- [GOO79] Goodenough, J. (1979), "A Survey of Program Testing Issues," Research Directions in Software Technology, The MIT Press.
- [GOO80] Goodenough, J. (1980), "The ADA Compiler Validation Capability," Proc. ACM SIGPLAN Symposium on the ADA Programming Language, Boston, MA.
- [GRI78] Griffiths, S.N. (1978), "Design Methodologies -- A Comparison," Structured Analysis and Design, Vol. II, Infotech
- [HAL77] Halsted, M.H. (1977), Elements of Software Science, Elsevier North Holland Publishing Co., New York.
- [HAN76] Han, Y. (1976), "A Systematic Study of Computer System Reliability," Ph.D. Thesis, University of California, Berkeley.
- [HO78] Ho, Siu-Bun Franklin, "A Systematic Approach to the Development and Validation of Software for Critical Applications," Ph.D. Dissertation, Univ. of California, Berkeley, 1978.
- [HOW80] Howden, W. (1980), "Functional Program Testing," IEEE Transactions on Software Engineering, Vol. SE-6, No. 2, pp. 162-169.
- [HUA75] Huang, J.C. (1975), "An Approach to Program Testing," Computing Surveys, Vol. 7, No. 3, pp. 113-128.

- [JEL72] Jelinski, Z. and Moranda, P. (1972), "Software Reliability Research," in Statistical Computer Performance Evaluation, W. Freiberger (Ed.), Academic Press, pp. 465-484.
- [JEL79] Jensen, R.W. and Tonies, C.C. (1979), Software Engineering, Prentice-Hall, Inc.
- [JON80] Jones, C.B. (1980), Software Development - A Rigorous Approach, Prentice-Hall International Series in Computer Science, London.
- [KAR75] Karlin, S. and Taylor, H.M. (1975), A First Course in Stochastic Processes, Academic Press, New York.
- [KLE76] Kleinrock, L. (1976), Queueing Systems, Vol. II: Computer Applications, John Wiley & Sons, Inc., New York.
- [KOB78] Kobayashi, H. (1978), Modeling and Analysis: An Introduction to System Performance Evaluation Methodology, Addison-Wesley.
- [LAN77] Landrault, C. and Laprie, J.S. (1977), "Reliability and Availability Modeling of Systems Featuring Hardware and Software Faults," Proc. 7th Annual International Conference on Fault-Tolerant Computing, Los Angeles, CA.
- [LEW64] Lewis, P.A.W. (1964), "Implications of a Failure Model for the Use and Maintenance of Computers," Journal of Applied Probability, Vol. 1, pp. 347-368.
- [LEW76] Lewis, P.A.W. and Shedler, G.S. (1976), "Statistical Analysis of Non-stationary Series of Events in a Data Base System," IBM Journal of Research and Development, Vol. 20, pp. 465-482.
- [LIP72] Lipow, M. (1972), Estimation of Software Package Residual Errors, TRW Software Series Report, TRW-SS-72-09, Redondo Beach, CA.
- [LIP73] Lipow, M. (1973), Maximum Likelihood Estimation of Parameters of a Software Time-to-Failure Distribution, TRW Systems Group Report, 2260.1.9-73B-15, Redondo Beach, CA.
- [LIP74] Lipow, M. (1974), "Some Variations of a Model for Software Time-to-Failure," TRW Systems Group, Correspondence ML-74-2260.1.9-21.

- [LIT73] Littlewood, B. and Verall, J.L. (1973), "A Bayesian Reliability Growth Model for Computer Software," Applied Statistics, Vol. 22, No. 3, pp. 332-346.
- [LIT75] Littlewood, B. (1975), "A Reliability Model for Systems with Markov Structure," Applied Statistics, Vol. 24, No. 2, pp. 172-177.
- [LIT76] Littlewood, B. (1976), "A Semi-Markov Model for Software Reliability with Failure Costs," Proc. MRI Symposium on Software Engineering, New York, pp. 281-300.
- [LIT80] Littlewood, B. (1980), "Theories of Software Reliability: How Good Are They and How Can They Be Improved?" IEEE Transactions on Software Engineering, Vol. SE-6, No. 5.
- [LLO79] Lloyd, D.K. and M. Lipow (1979), "Reliability: Management, Methods, and Mathematics," published by the authors, Redondo Beach, CA.
- [LON80] Longbottom, R., Computer System Reliability, John Wiley and Sons, Inc.
- [MAG52] Maguire, B.A., Pearson, E.S., and Wynn, A.H.A. (1952), "The Time Intervals Between Industrial Accidents," Biometrika, Vol. 39, pp. 168-180.
- [MAN74] Mann, N.R., Schafer, R.E., and Singpurwalla, N.D. (1974), Methods for Statistical Analysis of Reliability and Life Data, John Wiley & Sons.
- [MEY78] Meyer, J.F. (1978), "On Evaluating the Performability of Degradable Computing Systems," Proc. 8th Annual International Conference on Fault-Tolerant Computing, pp. 44-49.
- [MIL76] Miller, D.R. (1976), "Order Statistics, Poisson Processes and Repairable Systems," Journal of Applied Probability, Vol. 13, pp. 519-529.
- [MIL72] Mills, H.D. (1972), On the Statistical Validation of Computer Programs, IBM Federal Systems Division, Gaithersburg, MD, Report 72-6015.
- [MIY75] Miyamoto, I. (1975), "Software Reliability in On-Line Real Time Environment," Proc. International Conference on Reliable Software, Los Angeles, CA, pp. 194-203.

- [MOE76] Moeller, S.K. (1976), "The Rasch-Weibull Process," Scandinavian Journal of Statistics, Vol. 3, pp. 107-115.
- [MOR75a] Moranda, P.B. (1975), "Prediction of Software Reliability During Debugging," Proc. Annual Reliability and Maintainability Symposium, Washington, DC, pp. 327-332.
- [MOR75b] Moranda, P.B. (1975), "A Comparison of Software Error-Rate Models," Proc. 1975 Texas Conference on Computing, pp. 2A6.1-6.9.
- [MOR75c] Moranda, P.B. (1975), "Probability-Based Models for the Failures During Burn-In Phase," Joint National Meeting ORSA/TIMS, Las Vegas, NV.
- [MOR81] Moranda, P.B. (1981), "Event-Altered Rate Models for General Reliability Analysis," IEEE Transactions on Reliability, Vol. R-30, No. 2.
- [MOR82] Moranda, P.B. (1982), Private communication.
- [MUS75] Musa, J.D. (1975), "A Theory of Software Reliability and Its Application," IEEE Transactions on Software Engineering, Vol. SE-1, No. 3, pp. 312-327.
- [MUS80] Musa, J.D., Software Reliability Data, DACS, RADC, New York.
- [MUT77] Muth, E.J. (1977), Transform Method with Applications to Engineering and Operations Research, Prentice-Hall.
- [MYE75] Myers, G.J. (1975), Reliable Software Through Composite Design, Petrocelli/Charter, New York.
- [MYE76] Myers, G.J. (1976), Software Reliability, Principles and Practices, John Wiley & Sons, New York.
- [MYE79] Myers, G.J. (1979), The Art of Software Testing, John Wiley and Sons, Inc.
- [NEL75] Nelson, E.C. (1975), Software Reliability, TRW Software Series, TRW-SS-75-05, Redondo Beach, CA.
- [NEL78] Nelson, E. (1978), "Estimating Software Reliability from Test Data," Microelectronics and Reliability, Vol. 17, pp. 67-74.

- [OKU78] Okumoto, K. and Goel, A.L. (1978), Classical and Bayesian Inference for the Software Imperfect Debugging Model, Syracuse University, Final Technical Report, RADC-TR-78-155, Vol. 2 (of 5) AD A057-871.
- [OKU78] Okumoto, K. and Goel, A.L. (1978), Availability Analysis of Software Systems Under Imperfect Maintenance, Syracuse University, Final Technical Report, RADC-TR-78-155, Vol. 3 (of 5), AD A057-872.
- [OKU78] Okumoto, K. and Goel, A.L. (1978), "Availability and Other Performance Measures of Software Systems Under Imperfect Maintenance," Proc. COMPSAC, 1978, pp. 66-71.
- [ORR77] Orr, K.T. (1977), "Using Structured Systems Design," Structured Systems Development, Yourdon Press.
- [ORR78] Orr, K.T. (1978), "Introducing Structured Systems Design, Structured Analysis and Design, Vol. II, Infotech International Limited.
- [PIE76] Pierskalla, W.P. and Voelker, J.A. (1976), "A Survey of Maintenance Models: The Control and Surveillance of Deteriorating Systems," Naval Research Logistics Quarterly, Vol. 23, No. 3, pp. 353-388.
- [PRO63] Proschan, F. (1963), "Theoretical Explanation of Observed Decreasing Failure Rate," Technometrics, Vol. 5, No. 3, pp. 375-383.
- [PYK61] Pyke, R. (1961), "Markov Renewal Processes: Definitions and Preliminary Properties," Annals of Mathematical Statistics, Vol. 32, pp. 1231-1242.
- [RAH78] Raha, D., and Silva, N. (1978), "Digital Communication Systems - Reliability Trends," Proc. 1979 Annual Reliability and Maintenance Symposium, pp. 452-459.
- [RAM82] Ramamoorthy, C.V. and Bastani, F.B. (1981), "Software Reliability - Status and Perspectives," IEEE Transactions on Software Engineering, Vol. SE-8, No. 4.
- [REY79] Reynolds, J. (1981), The Craft of Programming, Prentice-Hall.
- [RIC81] Richardson, D.J. (1981), "A Partition Analysis Method to Demonstrate Program Reliability," Ph.D. Dissertation, University of Massachusetts, Amherst.

- [ROH76] Rohatgi, V.K. (1976), An Introduction to Probability Theory and Mathematical Statistics, John Wiley and Sons, Inc.
- [ROS76] Ross, S.M. (1976), Applied Probability Models with Optimization Applications, Holden-Day.
- [ROU73] Roussas, G.G. (1973), A First Course in Mathematical Statistics, Addison Wesley Publishing Co.
- [RUD79] Rudkin, R.I., and Shere, K.D. (1979), "Structured Decomposition Diagram: A New Technique for Systems Analysis," Datamation.
- [RYE77] Rye, P. et al. (1977), Software Systems Development: A CSDL Project History, The Charles Start Draper Laboratory, Inc., Final Technical Report, RADC-TR-77-213, AD A042-186.
- [SCH79] Schafer, R.E., et al. (1979), Validation of Software Reliability Models, Huges Aircraft Co., Final Technical Report, RADC-TR-78-147, AD A072-113.
- [SCH73] Schick, G.J. and Wolverton, R.W. (1973), "Assessment of Software Reliability," 11th Annual Meeting of the German Operations Research Society, DGOR, Hamburg, Germany; also in Proc. Operations Research, Physica-Verlag, Wurzburg-Wien, pp. 395-422.
- [SCH78] Schick, G.J. and Wolverton, R.W. (1978), "An Analysis of Computing Software Reliability Models," IEEE Trans. on Software Engineering, Vol. SE-4, No. 2, pp. 104-120.
- [SCH72] Schneidewind, N.F. (1972), "An Approach to Software Reliability Prediction and Quality Control," Proc. AFIPS Fall Joint Computer Conference, Vol. 41, Part II, pp. 837-838.
- [SCH75] Schneidewind, N.F. (1975), "Analysis of Error Processes in Computer Software," Proc. International Conference on Reliable Software, Los Angeles, CA, pp. 337-346.
- [SHO72] Shooman, M.L. (1972), "Probabilistic Models for Software Reliability Prediction," Statistical Computer Performance Evaluation, W. Freiburger (Ed.), Academic Press, pp. 485-502.

- [SHO73] Shooman, M. (1973), "Operational Testing and Software Reliability Estimation During Program Development," 1973 IEEE Symposium on Computer Software Reliability, New York City, April 30-May 2.
- [SHO75] Shooman, M.L. (1975), "Software Reliability Measurement and Models," Proc. 1975 Annual Reliability and Maintainability Symposium, pp. 485-491.
- [SHO76] Shooman, M.L. (1976), "Structural Models for Software Reliability and Prediction," Proc. 2nd International Conference on Software Engineering, pp. 268-273.
- [SNY75] Snyder, D.L. (1975), Random Point Processes, John Wiley and Sons, Inc.
- [SUK76] Sukert, A.N. (1976), A Software Reliability Modeling Study, In-house Technical Report, RADC-TR-76-247, AD A030-437.
- [SUK77] Sukert, A.N. (1977), "An Investigation of Software Reliability Models," Proc. Annual Reliability and Maintainability Symposium, Philadelphia, PA, pp. 478-484.
- [SUK78] Sukert, A.N. and Goel, A.L. (1978), "Error Modeling Applications in Software Quality Assurance," Proc. Software Quality and Assurance Workshop, San Diego, CA, pp. 33-38.
- [SUK80] Sukert, A.N. and Goel, A.L. (1980), "A Guidebook for Software Reliability Assessment," Proc. Annual Reliability and Maintainability Symposium, San Francisco, CA, pp. 188-190.
- [TAI80] Tai, K.C. (1980), "Program Testing Complexity and Test Criteria," IEEE Transactions on Software Engineering, Vol. SE-6, No. 6.
- [THA76] Thayer, T.A., Lipow, M., and Nelson, E.C. (1976), Software Reliability Study, TRW Defense and Space Systems Group, Final Technical Report, RADC-TR-76-238, AD A030-798.
- [THA78] Thayer, T.A., Lipow, M., and Nelson, E.C. (1978), Software Reliability, North-Holland, Amsterdam.
- [TRI75] Trivedi, A.K. (1975), Computer Software Reliability: Many-State Markov Modelling Techniques, Ph.D. Dissertation, Dept. of Electrical Engr., Polytechnic Institute of New York, NY.

- [TRI74] Trivedi, A.K. and Shooman, M.L. (1974), "A Markov Model for the Evaluation of Computer Software Performance," Research Report, Polytechnic Institute EE/EP, 74-011-EER 110.
- [TRI75] Trivedi, A.K. and Shooman, M.L. (1975), Computer Software Reliability: Many State Markov Modelling Techniques, Polytechnic Institute of New York, Interim Report, RADC-TR-75-169, AD 014-824.
- [WAG73] Wagoner, W.L. (1973), The Final Report on a Software Reliability Measurement Study, Technology Division, The Aerospace Corp., Report No. TOR-0074 (4112)-1, El Segundo, CA.
- [WEY80] Weyuker, E. and T. Ostrand (1980), "Theories of Program Testing and the Application of Revealing Subdomains," IEEE Transactions on Software Engineering, Vol. SE-6, No. 3.
- [WIL77] Willman, H.E., Jr., et al. (1977), Software Systems Reliability: A Raytheon Project History, Raytheon Co., Final Technical Report, RADC-TR-77-188.
- [WUL75] Wulf, W.A. (1975), "Reliable Hardware/Software Architecture," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, pp. 233-240.
- [YAU79] Yau, S.S. and MacGregor, T.E. (1979), On Software Reliability Modeling, Interim Report, Northwestern University, RADC-TR-79-129.
- [YOU72] Yourdon, E. (1972), "Reliability Measurements for Third Generation Computer Systems," Proc. Annual Reliability and Maintainability Symposium, pp. 174-183.
- [YOU75] Yourdon, E. (1975), Techniques of Program Structure and Design, Prentice-Hall, Inc., Englewood Cliffs, NJ.
- [ZEI81] Zeil, S. and White, L. (1981), "Sufficient Test Sets for Path Analysis Testing Strategies," Proc. 5th International Conference on Software Engineering, San Diego, CA.
- [ZIS75] Zislis, P. (1973), "Semantic Decomposition of Computer Programs: An Aid to Program Testing," Acta-Informatica 4.
- [ZWE81] Zweben, S. and Haley, A. (1981), "An Approach to Reliable Integration Testing," Technical Report No. 81-5, Ohio State University.

Appendix A

SOFTWARE ERRORS: THEIR SOURCES AND CLASSIFICATION

A.1 Sources of Software Errors

Software (also called program) is essentially an instrument for transforming a discrete set of inputs into a discrete set of outputs (see Figure A.1). It comprises of a set of coded statements whose function may basically be one of the following:

1. Evaluate an expression and store the result in a temporary or permanent location.
2. Decide which statement to execute next.
3. Perform input/output operations.

Since, to a large extent software is produced by humans, the finished software product is often imperfect. It is imperfect in the sense that a discrepancy exists between what the software can do versus what the user or the computing environment wants it to do. The computing environment refers to the physical machine, operating system, compiler and translators, utilities, etc. These discrepancies are what we call software errors (see Figure A.2). Basically, the software errors can be attributed to the following:

1. Ignorance of the user requirements,
2. Ignorance of the rules of the computing environment, and

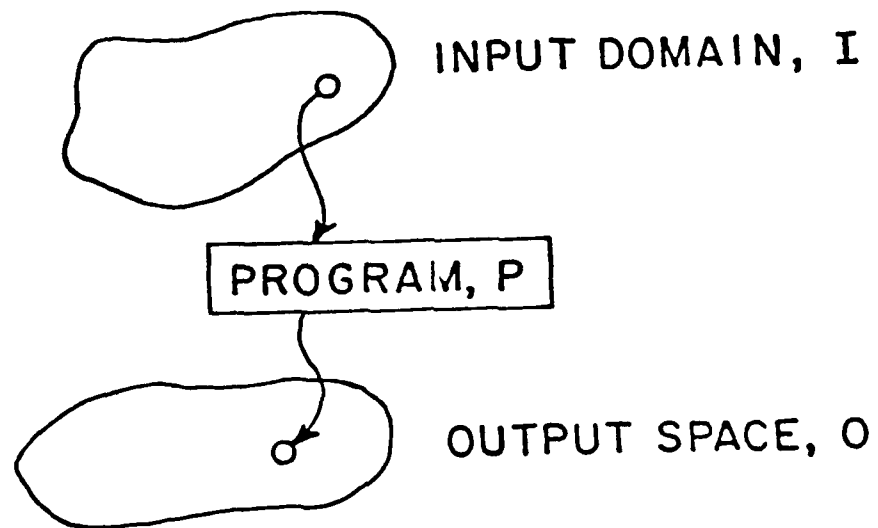


FIG. A.1. FUNCTIONAL VIEW OF SOFTWARE

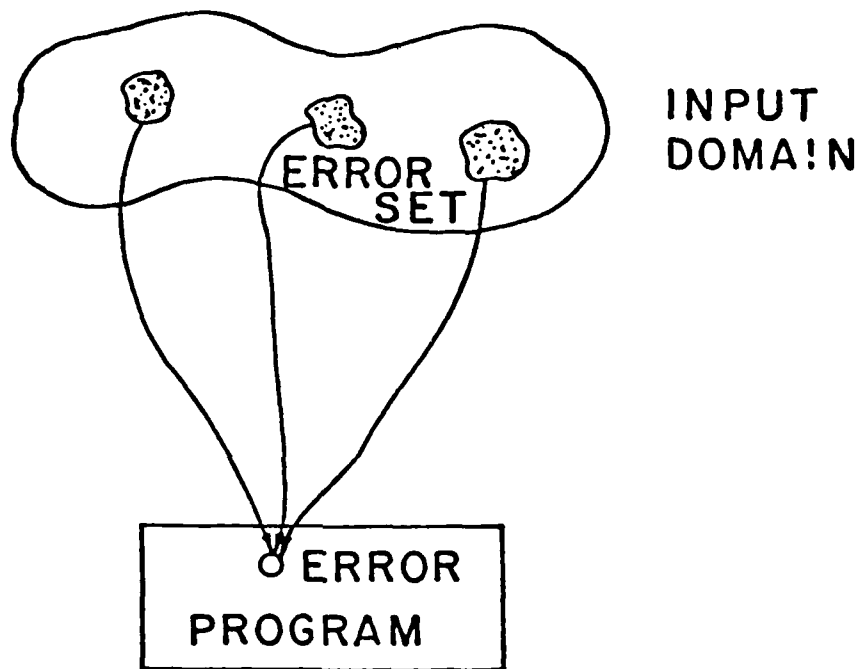


FIG. A.2. SOFTWARE ERROR

3. Poor communication of software requirements between the user and the programmer or poor documentation of the software by the programmer.

The fact of the matter is even if we know that software contains errors, we may not know with certainty the exact identity of these errors.

Currently, there are two major paths one can follow to expose software errors:

1. Program proving, and
2. Program testing.

Program proving is more formal and mathematical while program testing is more practical and still remains to be heuristic in its approach. The approach in program proving is the construction of a finite sequence of logical statements ending in the statement (usually the output specification statement) to be proved. Each of the logical statements is an axiom or is a statement derived from earlier statements by the application of an inference rule. Program proving making use of inference rules is known as the Inductive Assertion Method. This method was mainly popularized by Floyd, Hoare, Dijkstra and recently Reynolds. Other work on program proving is the work on the Symbolic Execution Method. This method is the basis of some automatic program verifiers. Despite the formalism

and mathematical exactness of program proving, it is still an imperfect tool for verifying program correctness. Gerhart and Yelowitz [GER 76] showed several programs which were proven to be correct but still contained errors. The errors were due to failures in defining what exactly to prove and were not failures of the mechanics of the proof itself.

Program testing is the symbolic or physical execution of a set of test cases with the intent of exposing embedded errors (if any) in the program. Like program proving, program testing remains an imperfect tool for verifying program correctness. A given testing strategy is good for exposing certain kinds of errors but not all possible kinds of errors in a program. An advantage of testing is that it provides accurate information about a program's actual behavior in its actual computing environment; proving is limited to conclusions about the program's behavior in a postulated environment.

Neither proving nor testing can, in practice, guarantee complete confidence on the correctness of programs. Each has its pluses and minuses. They should not be viewed as competing tools. They are, in fact, complementary methods for decreasing the likelihood of program failure [GOO 77].

A.1. SOFTWARE ERROR CLASSIFICATION

A systematic study of software errors in a program requires knowing what specifically these errors are and knowing which tool(s) to use to expose particular types of software errors. Software errors can be grouped as syntax, semantic, runtime, specification and performance errors.

A.2.1 Syntax Errors

These errors are due to discrepancies between the program code and the syntax rules governing the parser or lexical analyzer of a program translator. These are the easiest errors to detect. They can be detected by visual inspection of the code or can be detected mechanically during the program compilation process. Experienced programmers rarely commit syntax errors.

A.2.2 Semantic Errors

These errors are due to discrepancies between the program code and what the semantic analyzer of the computing environment accepts. Among the popular kinds of semantic errors are typechecking errors and implementation restriction errors. Again, they may be detected by the semantic analyzer of a program translator or by visual inspection.

Syntax and semantic errors are detected during the compilation stage of a program. A program having syntax and/or semantic errors cannot be executed. Syntax and semantic errors are mainly due to the ignorance/negligence on the part of the programmer about the restrictions and limitations of the language (s)he is using.

A.2.3 Runtime Errors

As the name implies, runtime errors occur during the actual running of a program. They may be further classified into three categories:

Domain errors

A domain error occurs whenever the value of a program variable exceeds its declared range or exceeds the physical limits of the hardware representing the variable. The declared range of a variable is done implicitly or explicitly. FORTRAN, for example, assigns types to variables based on the variable name or based on a declaration statement. PASCAL requires all variables to be explicitly declared in a declaration statement. PASCAL has facilities to declare ranges by enumeration and/or subsets of numeric domains.

Some program translators produce runtime code for checking certain types of domain errors. Some have built-in recovery features for domain errors (e.g. PL/1, COBOL) and others (e.g. FORTRAN) simply abort execution upon the occurrence of a domain error. Certain compilers, like

PASCAL, automatically check for values outside a declared range.

Domain errors are a serious matter because

- a) program execution is aborted, and/or
- b) program results are incorrect.

Execution abortion may be fatal especially in real-time systems. Despite their seriousness, domain errors have never been formally and extensively studied in the literature. This is because detection of domain errors can be very difficult. They require exact specification of the ranges of the input variables. Also, the test values required to expose these errors may occur at the input domain's boundary or inside the input domain itself.

Computational errors

Computational errors, sometimes known as logic errors, result whenever the program results in an incorrect output. The incorrect output may be due to a wrong formula, an incorrect control flow, assignment to a wrong variable, incorrect parameter passing, etc.

It is not possible to generate runtime code to detect computational errors during program execution. This is because computational errors are really discrepancies between the program's output and the program's specifications.

Computational errors due to incorrect program constructs and statements may be detected by any of the structure dependent or structure independent testing techniques. However, none of these tools can guarantee total absence of these types of computational errors in a program. Computational errors due to missing program constructs and statements may be detected by any of the structure independent testing techniques. Again, none of these tools can guarantee total absence of computational errors due to missing paths.

Non-Termination errors

Non-termination error is simply the failure of a program to terminate in finite time without outside intervention. The most common cause of non-termination errors is when the program runs into an infinite loop. Non-termination can also occur if a set of concurrent programs falls into a dead lock.

Infinite loops are detected by simply executing each of the loops in a program. However, this strategy may not guarantee total absence of infinite loops. Some infinite loops may only occur if certain program variables achieve certain values. Program proving may also be used on certain programs to expose infinite loops. The problem of program non-termination in general is still an unsolved problem.

A.2.4 Specification Errors

Specification errors result whenever there exists a discrepancy between the statement of specifications and the statement of user requirements. A requirements error exists whenever there is a discrepancy between the statement of user requirements and the real user requirements.

Presently, detection of specification errors such as:

1. Incomplete specifications,
2. Inconsistent specifications, and
3. Ambiguous specifications,

remains an informal process. This is mainly due to the nonexistence of a specification language powerful enough to translate the user requirements into clear, complete and consistent terms.

A testing tool to detect specification errors is yet to be developed.

A.2.5 Performance Errors

Performance errors exist whenever a discrepancy exists between the actual performance (efficiency) of the programs and its desired or specified performance. Program performance may be measured in a number of ways:

1. Response time
2. Elapsed time
3. Memory space usage
4. Working set requirement, etc.

The actual measurement of the above measures of program performance can be a very difficult process. Program complexity theory tries to estimate bounds on the running time of certain program algorithms. Statistical analysis and simulation can also be employed to estimate the above performance variables. However, use of these tools can be very expensive and time consuming.

A performance testing tool that is economical (time-wise and costwise) to use is yet to be developed.

The most expensive kind of software errors to eliminate are those which are not discovered until late in the software development, such as when the software becomes operational. These are known as persistent software errors. Glass [GLA81] reported that persistent software errors are mostly due to the failure of the problem solution (i.e. the program) to match the complexity of the problem to be solved (i.e. the user requirements). Examples of such errors are computational errors due to missing or insufficient predicates and failure to reset a variable to some baseline value after its use in a functional logic segment. The solution to this software problem is beyond the current state-of-the-art.

Appendix B

BASIC RELIABILITY CONCEPTS

In this appendix we present some of the basic concepts from reliability theory that are useful in understanding the software reliability models. For a detailed discussion of this material, the reader should refer to a standard book such as Barlow and Proschan [BAR75] or Mann et al [MAN74].

In the following we will use the random variable X to denote the time to failure and x to denote a realization of X .

Cumulative Distribution Function (cdf)

The cdf of a random variable X is denoted by $F(x)$ and represents the probability of X being less than or equal to x , i.e.,

$$F(x) = P(X \leq x)$$

Probability Density Function (pdf)

The pdf of X is denoted by $f(x)$ and is defined as follows:

$$f(x) = \frac{d}{dx} F(x)$$

Reliability

The term reliability refers to the probability of failure free operation. Specifically, reliability $R(x)$ is defined as follows

$$R(x) = P(\text{no failure by } x)$$

or

$$R(x) = P(X \geq x)$$

This is an important and commonly used measure of a software system.

Hazard Function or Failure Rate

This is a very fundamental term in software (or hardware) reliability modelling work. The hazard function is defined as

$$z(x) = \frac{f(x)}{1 - F(x)}$$

It is a measure of the instantaneous speed of failure.

Reliability in Terms of Hazard Function

Given the hazard function, the software reliability can be computed from the following relationship.

$$R(x) = e^{-\int_0^x z(u) du}$$

Mean Time to Failure (MTTF)

This measure represents the expected value of the time to next failure and is a very commonly used measure of software quality. It can be computed by any of the following formulae

$$MTTF = \int_0^{\infty} xf(x) dx$$

$$MTTF = \int_0^{\infty} R(x) dx$$

Appendix C

DETAILS OF TIMES BETWEEN FAILURES (TBF) MODELS

This appendix contains theoretical and computational details of most of the TBF models discussed in Section 3. The following information is provided for each model:

- . Model Assumptions
- . Basic Formulae
- . Parametric Estimation
- . Performance Measures
- . Data Requirements
- . Model Applicability
- . Relevant References

C-1 Jelinski and Moranda De-eutrophication Model (Model TBF1)

1. Assumptions

- a. Initial fault content
 - . An unknown fixed constant N
- b. Independence of faults
 - . Each fault in the program is independent of other faults and each of them is equally likely to cause a failure during testing. Times between occurrences of faults are independent of each other.
- c. Fault removal process
 - . A detected fault is removed with certainty at the end of each testing interval
 - . Only one fault is removed during each testing interval
 - . The fault removal time is negligible
 - . No new faults are introduced during the fault removal process.
- d. Hazard function
 - . The software failure rate of the hazard function during a failure interval is constant and is proportional to the current fault content of the tested program. Thus, during the i th testing interval,

$$z(t_i) = \phi\{N - (i-1)\},$$

where ϕ is a proportionality constant, and N is the initial number of faults in the program.

2. Basic Formulae

The time between the (i-1)st and the ith failures T_i , is distributed exponentially with parameter $\phi[N - (i-1)]$.

$$\text{pdf of } T_i: f(t_i) = \phi[N - (i-1)] \exp[-\phi\{N - (i-1)\}t_i]$$

$$\text{cdf of } T_i: F(t_i) = 1 - \exp[-\phi\{N - (i-1)\}t_i]$$

Plots of pdf and cdf for $N = 100$ and $\phi = .01$ are shown in Figure D-1.

3. Estimation of Parameters and Related Results

A series of n failures is observed with interfailure times as t_1, t_2, \dots, t_n . Usually, the method of maximum likelihood is used to estimate the parameters N and ϕ as shown below.

The likelihood function of N and ϕ is

$$L(N, \phi | t_1, t_2, \dots, t_n) = \prod_{i=1}^n \phi[N - (i-1)] \exp[-\phi\{N - (i-1)\}t_i]$$

The maximum likelihood estimates (mle's) of N and ϕ are obtained by solving the following pair of equations simultaneously.

$$\sum_{i=1}^n \frac{1}{N - (i-1)} - \sum_{i=1}^n \phi t_i = 0$$

and

$$\frac{n}{\phi} - \sum_{i=1}^n [N - (i-1)] t_i = 0$$

The above two equations yield

$$\sum_{i=1}^n \frac{1}{N - (i-1)} = \frac{n}{N - \frac{1}{T} \sum_{i=1}^n (i-1)t_i},$$

$$\text{where } T = \sum_{i=1}^n t_i$$

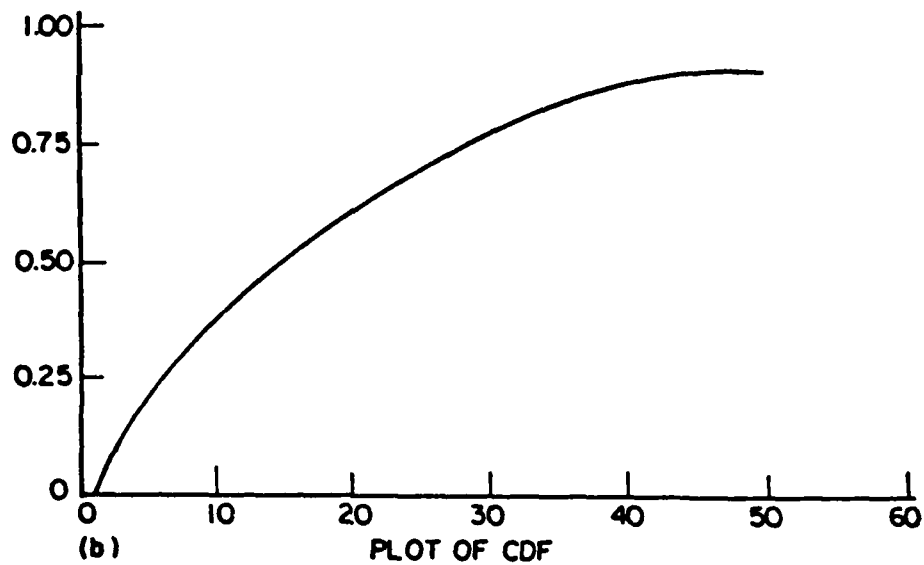
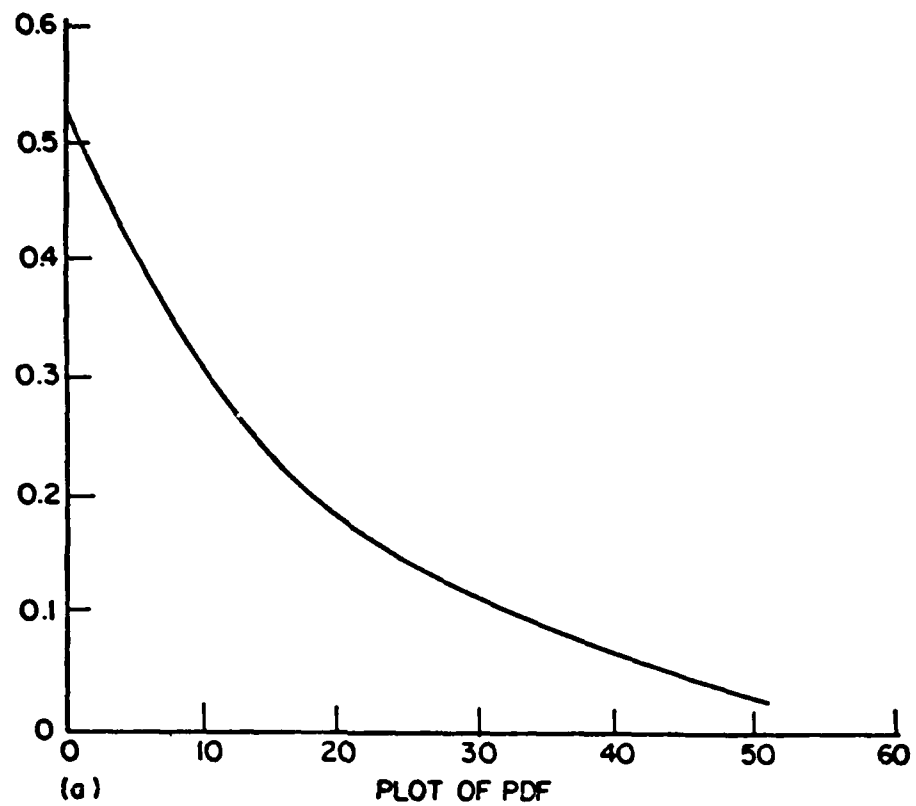


Figure C.1. Plots of PDF and CDF for Model TBF1 ($N = 100$, $\phi = 0.01$).

We can solve the above equation numerically to find \hat{N} and then obtain $\hat{\phi}$ from the following equation

$$\hat{\phi} = \frac{n}{\hat{N}T - \sum_{i=1}^n (i-1)t_i}$$

Using the asymptotic properties of the maximum likelihood estimators, it can be shown that

$$\text{Var}(\hat{N}) = \frac{n}{\phi^2} \cdot \frac{1}{\det A}$$

$$\text{Var}(\hat{\phi}) = \sum \left(\frac{1}{N-i+1} \right)^2 \cdot \frac{1}{\det A}$$

$$\text{Cov}(\hat{N}, \hat{\phi}) = -\frac{T}{\det A}$$

$$\rho_{\hat{N}, \hat{\phi}} = -\frac{T\phi}{\sqrt{n} \cdot \sqrt{\Sigma_2}}$$

where

$$\det A = \frac{n}{\phi^2} \sum_{i=1}^n \left(\frac{1}{N-i+1} \right)^2 - T^2,$$

and

$$\Sigma_2 = \sum_{i=1}^n \frac{1}{(N-i+1)^2}$$

4. Performance Measures

- Reliability at time t after the n th failure is

$$R_n(t) \equiv P(T_{n+1} > t) = \exp[-\phi \{N-n\}t]$$

- Mean Time to Failure (MTTF) after n failures is

$$\text{MTTF}_n = \int_0^{\infty} R_n(t) dt = \frac{1}{\phi [N-n]}$$

Plots of $R_{50}(t)$ and of MTTF after 1, 2, ..., 25 failures are shown in Figure C-2.

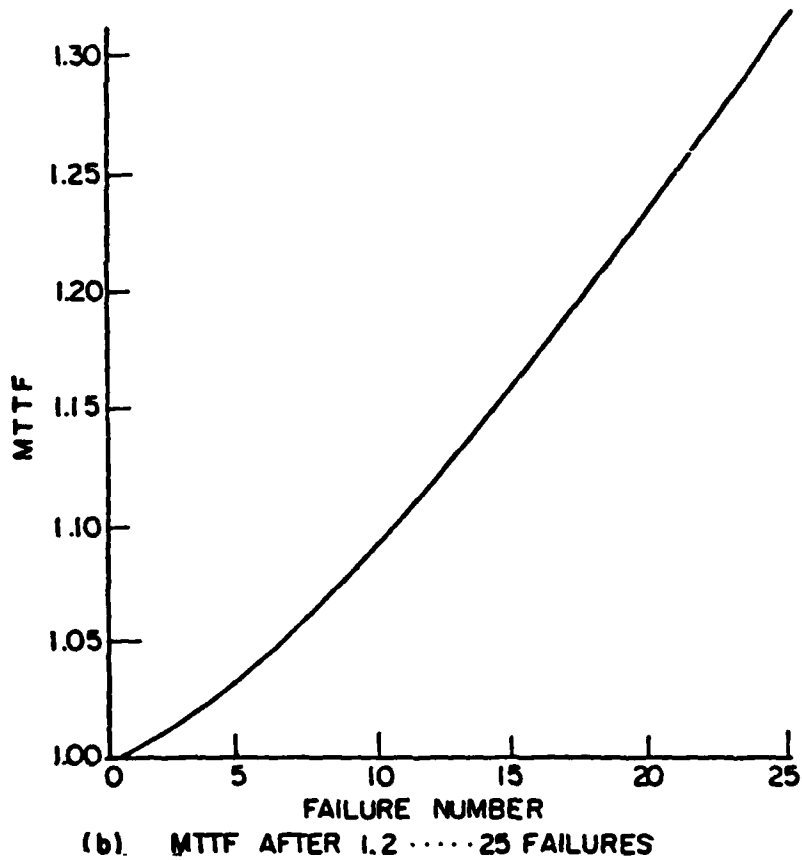
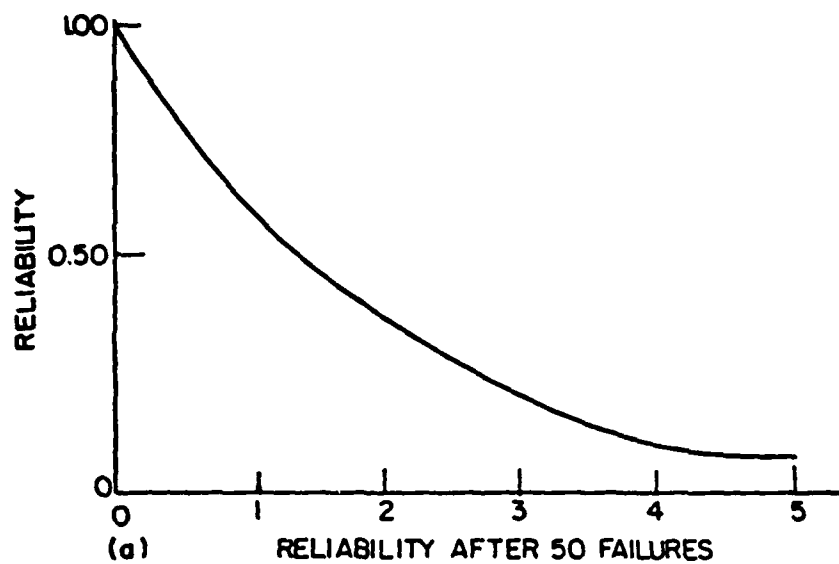


Figure C.2. Plots of Reliability and MTTF for Model TBF1 ($N = 100$, $\phi = 0.01$).

Using the variance-covariance of $\hat{N}, \hat{\phi}$, it can be shown that

$$\text{Variance}(R_n(t)) = \frac{e^{-2\phi_1(N-n)t}}{\det A} \{nt^2 - 2(N-n)Tt^2\phi + (N-n)^2t^2\Sigma_2\}$$

$$\text{Variance}(MTTF_n) = \frac{1}{\det A} \left[\frac{n}{\phi^2} v_1^2 - 2Tv_1v_2 + v_2^2\Sigma_2 \right],$$

where

$$v_1 = -\frac{1}{(N-n)^2\phi}$$

$$v_2 = -\frac{1}{(N-n)\phi^2}, \text{ and}$$

Σ_2 and $\det A$ are as defined before.

All the above quantities can be computed by replacing N and ϕ by \hat{N} and $\hat{\phi}$, respectively.

5. Data Requirements

Data on times between failures, t_1, t_2, \dots, t_n are required to estimate the parameters N and ϕ (see #3 above).

6. Model Applicability

If the underlying assumptions are satisfied, this model can be employed during the system integration testing phase.

7. Relevant References

[JEL72] was the original paper in which this model was introduced. Further discussion and some additional details are in [MOR75, MOR81]. Computations of various relevant quantities and appropriate confidence bounds are discussed in [GOE79a, GOE80].

C-2. Schick-Wolverton Linear Model (Model TBF2)

1. Assumptions

a. Initial fault content

- . An unknown fixed constant N

b. Independence of faults

- . Each fault in the program is independent of other faults and each of them is equally likely to cause a failure during testing. Times between occurrences of faults are independent of each other.

c. Fault removal process

- . A detected fault is removed with certainty at the end of the debugging interval
- . Only one fault is removed during each testing interval
- . The fault removal time is negligible
- . No new faults are introduced during the fault removal process

d. Hazard function

- . The software failure rate or the hazard function, at any time is proportional to the current fault content of the program and to the time elapsed since the last failure. Thus, the hazard function between the (i-1)st and the ith failures is given by

$$z(t_i) = \phi[N - (i-1)]t_i$$

where ϕ is a proportionality constant, N is the initial fault content, and t_i is the test time since the (i-1)st failure.

2. Basic Formulae

The time between the (i-1)st and the ith failures T_i has a Rayleigh distribution with scale parameter equal to $\{\phi[N - (i-1)]/2\}^{1/2}$. Note that Rayleigh is a special of the Weibull distribution with shape parameter equal to 2.

$$\text{pdf of } T_i: f(t_i) = \phi[N - (i-1)]t_i \exp[-\phi[N - (i-1)]t_i^2/2]$$

$$\begin{aligned} \text{cdf of } T_i: F(t_i) &= 1 - \exp[-\phi[N - (i-1)] \int_0^{t_i} t \, dt] \\ &= 1 - \exp[-\phi[N - (i-1)]t_i^2/2] \end{aligned}$$

The pdf of each failure interval is a different Rayleigh distribution and each is an Increasing Failure Rate (IFR) distribution [BAR75,GOE80]. Plots of pdf and cdf for $N = 150$, $\phi = .02$ are shown in Figure C-3.

3. Estimation of Parameters

A series of n failures is observed with interfailure times as t_1, t_2, \dots, t_n . Usually the method of maximum likelihood is used to estimate the parameters N and ϕ as shown below.

The likelihood function of N and ϕ is

$$L(N, \phi | t_1, t_2, \dots, t_n) = \prod_{i=1}^n \phi [N - (i-1)]t_i \exp[-\phi [N - (i-1)]t_i^2/2]$$

The maximum likelihood estimates of parameters N and ϕ are obtained by solving the following pair of equations simultaneously.

$$\frac{2n}{\phi} - \sum_{i=1}^n (N - (i-1))t_i^2 = 0$$

and

$$\sum_{i=1}^n \frac{1}{N - (i-1)} - \frac{\phi}{2} \sum_{i=1}^n t_i^2 = 0$$

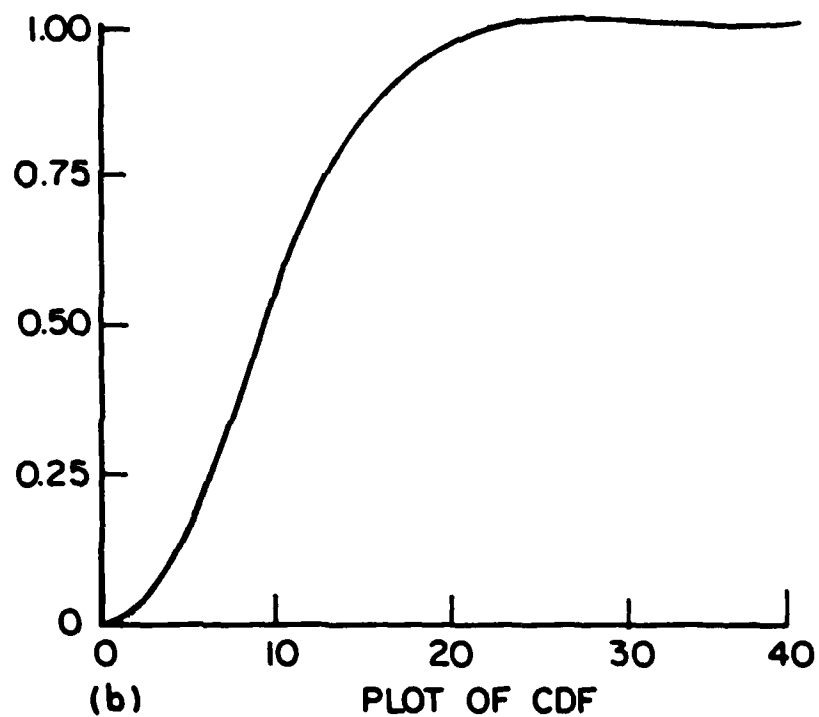
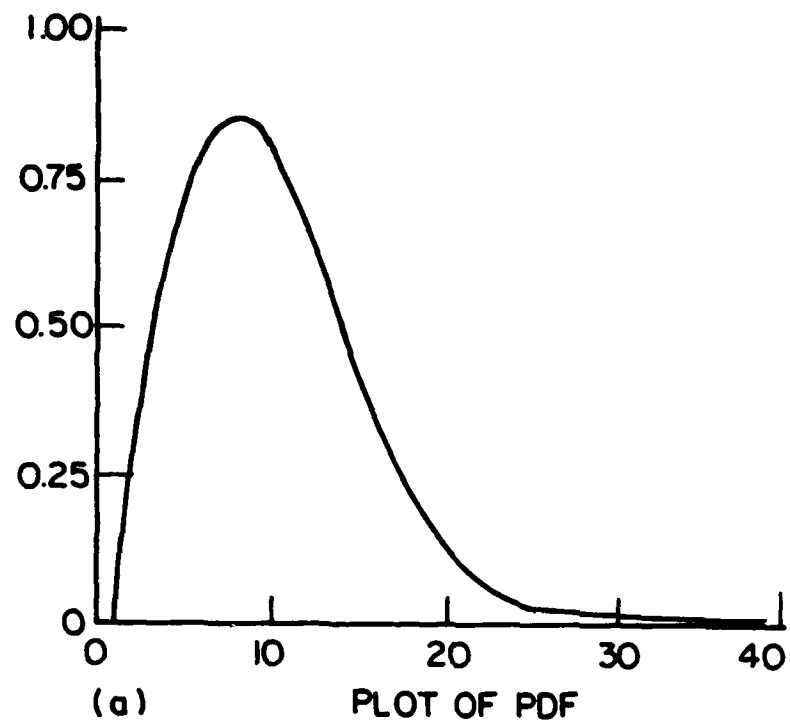


Fig. C.3 Plots of PDF and CDF for Model TBF2
($N = 150$, $\phi = 0.02$)

The above two equations yield

$$\sum_{i=1}^n \frac{1}{N-(i-1)} = \frac{\sum_{i=1}^n t_i^2}{N \sum_{i=1}^n t_i^2 - \sum_{i=1}^n (i-1)t_i^2}$$

We can solve the above equation numerically to find \hat{N} and then obtain $\hat{\phi}$ from the following equation

$$\hat{\phi} = \frac{2n}{\sum_{i=1}^n [\hat{N}-(i-1)]t_i^2} = \frac{2n}{\hat{N} \sum_{i=1}^n t_i^2 - \sum_{i=1}^n (i-1)t_i^2}$$

4. Performance Measures

Reliability at time t after the n th failure is

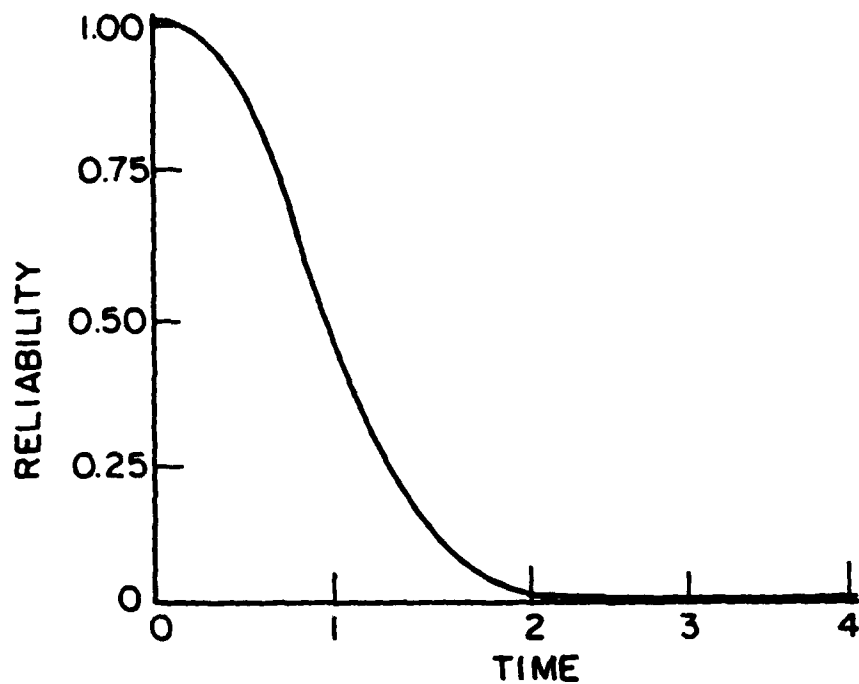
$$R_n(t) \equiv P(T_{n+1} > t) = \exp[-\phi \{N-n\} \frac{t^2}{2}]$$

Mean Time to Failure (MTTF) after n failures is

$$MTTF_n = \int_0^{\infty} R(t) dt = \int_0^{\infty} \exp[-\phi \{N-n\} \frac{t^2}{2}] dt$$

$$\text{or } MTTF_n = \left[\frac{\pi}{2} \cdot \frac{1}{\phi \{N-n\}} \right]^{1/2}$$

Plots of $R_{50}(t)$ and MTTF after 1,2,3,4, and 5 failures are shown in Figure C-4.



(a) RELIABILITY AFTER 50 FAILURES

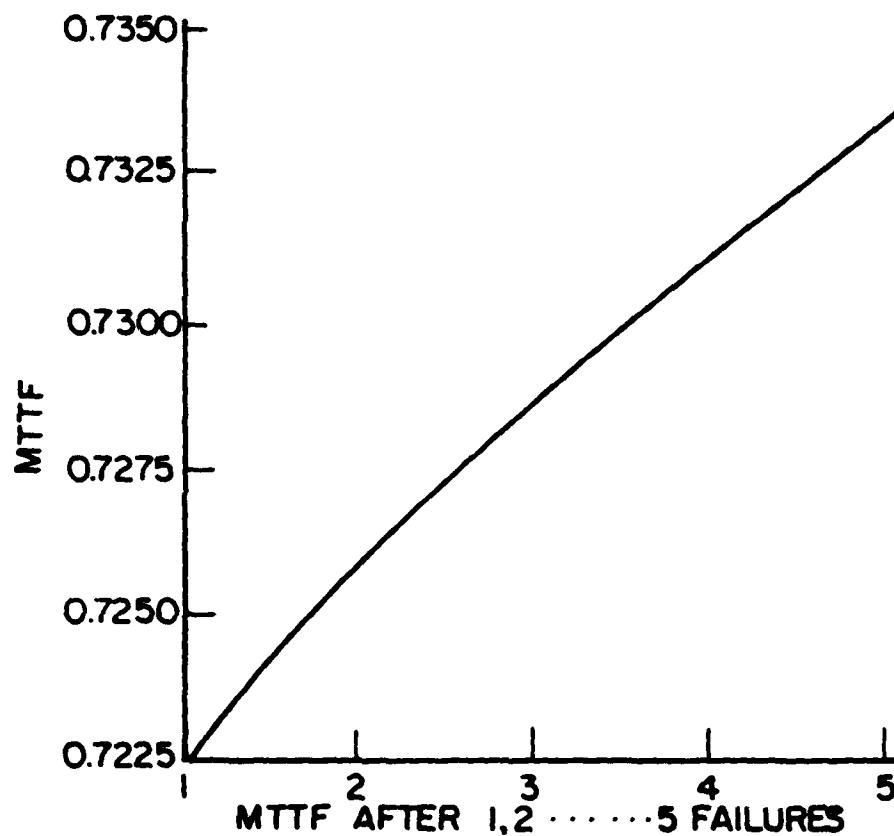


Figure C.4 Plots of Reliability and MTTF for Model TBF2
($N = 150$, $\phi = 0.02$)

5. Data Requirements

Data on times between failures t_1, t_2, \dots, t_n are required to estimate the parameters N and ϕ_2 (see #4 above).

6. Model Applicability

Here times between failures are assumed to follow IFR distributions and hence this model should be used only when the testing strategy justifies such increasing fault detection rate.

7. Relevant References

The model was first proposed by Schick and Wolverson [SCH72]. A comparative study with other models was reported in [SCH78]; however, some of the material in that paper could be misinterpreted as pointed out in [GOE80].

C-3. Geometric De-eutrophication Model (Model TBF4)

1. Assumptions

- a. Initial fault content:
 - . No specific value needs to be assumed
- b. Independence of fault:
 - . Fatal faults are independent of each other and each of them is equally likely to cause a failure during testing. Times between failures are independent of each other.
- c. Fault removal process:
 - . Testing is carried out until a fatal fault occurs and then the accumulated group of faults is removed along with the fatal fault.
 - . A fatal fault (possibly with other non-fatal faults) is removed with certainty at the end of each testing interval
 - . The fault removal time is negligible
 - . No new faults are introduced during the fault removal process
- d. Hazard function:
 - . The software failure rate or the hazard function during a testing interval is a constant but changes values at occurrences of failures. For the i th such interval, the hazard function is given by

$$z(t_i) = Dk^{i-1}$$

where D is the fault detection rate during the first interval and k is a constant, $0 < k < 1$.

2. Basic Formulae

Time between the (i-1)st and ith failures, T_i , is distributed exponentially with parameter Dk^{i-1}

$$\text{pdf of } T_i: f(t_i) = Dk^{i-1} \exp[-Dk^{i-1}t_i]$$

$$\text{cdf of } T_i: F(t_i) = 1 - \exp[-Dk^{i-1}t_i]$$

Plots of pdf and cdf for $D = 0.5$, $k = 0.95$ are shown in Figure D-3.

3. Estimation of Parameters and Related Results

A series of n failures is observed with interfailure times as t_1, t_2, \dots, t_n . Usually the method of maximum likelihood is used to estimate the parameter D and k as shown below.

The likelihood function of D, k is

$$L(D, k | t_1, t_2, \dots, t_n) = \prod_{i=1}^n Dk^{i-1} \exp[-Dk^{i-1}t_i]$$

The maximum likelihood estimators of parameters D and K can be obtained by solving the following pair of equations simultaneously.

$$\frac{n}{D} - \sum_{i=1}^n k^{i-1} t_i = 0$$

and

$$\frac{1}{k} \sum_{i=1}^n (i-1) - D \sum_{i=1}^n (i-1) k^{i-2} \cdot t_i = 0$$

The above two equations yield

$$\frac{\sum k^i t_i}{\sum k^{i-1} t_i} = \frac{n+1}{2}$$

from which we can find \hat{k} by solving the equation numerically, and then \hat{D} from the following equation

$$\hat{D} = \frac{n}{\sum_{i=1}^n \hat{k}^{i-1} t_i}$$

Using the asymptotic properties of the maximum likelihood estimators, it can be shown that

$$\text{Var}(\hat{D}) = D^2 \frac{2(2n-1)}{n(n+1)}$$

$$\text{Var}(\hat{k}) = k^2 \frac{12}{n(n^2-1)}$$

$$\text{Cov}(\hat{D}, \hat{k}) = -Dk \frac{6}{n(n+1)}$$

$$\rho_{\hat{D}, \hat{k}} = -\frac{\sqrt{3}}{2} \frac{(n-1)}{(2n-1)}$$

4. Performance Measures

- Reliability at time t after the n th failure is

$$R_n(t) \equiv P(T_{n+1} > t) = \exp[-Dk^n t]$$

- Mean Time to Failure (MTTF) after n failures is

$$\text{MTTF}_n = \int_0^{\infty} R_n(t) dt = \frac{1}{Dk^n}$$

Plots of $R_{25}(t)$ and MTTF after 1, 2, 11, 20 failures are shown in Figure D-3.

Using the variance-covariance of \hat{D} , \hat{k} it can be shown that

$$\text{Var}(\text{MTTF}_n) = \frac{2}{n(n^2-1)k^{2n}D^2} (2n^2 - 3n + 7)$$

5. Data Requirements

Data on times between fatal failures t_1, t_2, \dots, t_n are required to estimate the parameters D and k (see #4 above).

6. Model Applicability

This model can be used when testing continues until the occurrence of a fatal fault at which time the fatal and the non-fatal faults are removed.

7. Relevant references

The model was proposed by Moranda in [MOR75]. This model was used to analyze some failure data and to compare the results with some other models in [SUK76].

Appendix D
DETAILS OF FAILURE COUNT MODELS

In this appendix we present detailed technical material about the failure count models of Section 4. The following models are discussed:

- D-1 Goel-Okumoto Non-Homogeneous Poisson Process Model
(Model FC1)
- D-2 Goel Modified Non-Homogeneous Poisson Process Model
(Model FC3)
- D-3 Musa Execution Time Model
(Model FC4)
- D-4 Shooman Model (Model FC5)
- D-5 Geometric Poisson Model (Model FC6)
- D-6 Modified Jelinski-Moranda Model FC7)
- D-7 Modified Geometric De-Eutrophication Model
(Model FC8)
- D-8 Modified Schick-Wolverton Model (Model FC9)
- D-9 Generalized Poisson Model (Model FC10)
- D-10 IBM Binomial Model (Model FC11)
- D-11 IBM Poisson Model (Model FC12)

D-1 Goel-Okumoto Non-Homogeneous Poisson Process Model
(Model FC1)

1. Assumptions

a. Initial fault content:

- . Expected number of software faults to be eventually detected is an unknown fixed quantity
- . Actual number of faults to be observed is a random variable

b. Independence of faults:

- . Each failure is caused by one fault and each of them is equally likely to cause a failure during testing.
- . Number of software faults detected during non-overlapping testing intervals is independent of each other

c. Fault removal process:

- . Fault removal time is negligible
- . No new faults are introduced during the fault removal process.

d. Intensity function:

- . Expected number of software faults detected during $(t, t + \Delta t)$ is proportional to the expected number of software faults undetected by time t , i.e.

$$m(t + \Delta t) - m(t) = b\{a - m(t)\}\Delta t$$

where

$m(t)$ = expected number of software faults detected
by time t ,

a = expected number of software faults to be
eventually detected,

b = constant of proportionality.

. The intensity function or the fault-detection rate
 $\lambda(t)$ is a decreasing function of t and is given by

$$\lambda(t) \equiv m'(t) = b\{a - m(t)\} = abe^{-bt}$$

where a, b , and $m(t)$ are as defined above.

2. Basic Formulae

Expected number of software faults detected by time t
is given by

$$m(t) = a(1 - e^{-bt})$$

where a and b are as defined above.

The total number of software faults detected by time t ,
 $N(t)$, under the above assumptions is an NHPP with mean value
function $m(t)$ and intensity function $\lambda(t)$ as given above.

The distribution of $N(t)$, hence, is given by

$$P\{N(t) = y\} = \frac{\{m(t)\}^y}{y!} e^{-m(t)}, \quad y = 0, 1, 2, \dots$$

and

$$E[N(t)] = m(t) = a(1 - e^{-bt})$$

3. Estimation of Parameters

Suppose y_1, y_2, \dots, y_n are the cumulative number of
software faults detected by times t_1, t_2, \dots, t_n , respectively.
Then the likelihood function for (a, b) given the data pairs

$\{(y_i, t_i), i = 1, 2, \dots, n\}$ is

$$L(a, b | y_1, y_2, \dots, y_n, t_1, t_2, \dots, t_n) =$$

$$= \Pr\{N(t_1) = y_1, N(t_2) = y_2, \dots, N(t_n) = y_n\}$$

$$= \prod_{i=1}^n \frac{[m(t_i) - m(t_{i-1})]^{y_i - y_{i-1}} e^{-\{m(t_i) - m(t_{i-1})\}}}{(y_i - y_{i-1})!}$$

$$= \prod_{i=1}^n \frac{\{a(e^{-bt_{i-1}} - e^{-bt_i})\}^{y_i - y_{i-1}} e^{-a(1 - e^{-bt_n})}}{(y_i - y_{i-1})!}$$

MLE of parameters a and b can be obtained by solving the following pair of equations simultaneously

$$a(1 - e^{-bt_n}) = y_n$$

and

$$at_n e^{-bt_n} = \sum_{i=1}^n \frac{(y_i - y_{i-1})(t_i e^{-bt_i} - t_{i-1} e^{-bt_{i-1}})}{e^{-bt_{i-1}} - e^{-bt_i}}$$

The above two equations yield

$$\frac{y_n t_n e^{-bt_n}}{(1 - e^{-bt_n})} = \sum_{i=1}^n \frac{(y_i - y_{i-1})(t_i e^{-bt_i} - t_{i-1} e^{-bt_{i-1}})}{e^{-bt_{i-1}} - e^{-bt_i}}$$

from which we can find \hat{b} numerically and hence

$$\hat{a} = \frac{y_n}{(1 - e^{-\hat{b}t_n})}$$

4. Performance Measures

- Expected number of software faults detected by time t is given by

$$\hat{m}(t) = \hat{a}(1 - e^{-\hat{b}t})$$

- . Expected number of remaining faults in the s/w system at time t is given by

$$E[\hat{N}(t)] = \hat{a}e^{-\hat{b}t}$$

where $\hat{N}(t)$ = number of faults remaining in the system at time t.

- . Software reliability

Let X_k be the time between failures (k-1) and k and S_k be the time to k failures. Then it can be shown that the conditional reliability function of X_k , given $S_{k-1} = s$, is

$$\hat{R}_{X_k|S_{k-1}}(x|s) = \exp[-\hat{a}\{e^{-\hat{b}s} - e^{-\hat{b}(s+x)}\}]$$

5. Data Requirements

The data needed are lengths of testing intervals and number of failures in each interval. Before fitting the model, however, all data should be normalized to equal test intervals.

6. Model Applicability

Model can be used in a fairly general testing environment (see explanation of assumptions in Section 6.1).

7. Relevant References

The model was proposed in [GOE79a]. Detailed data analyses based on this model are given in [GOE82].

D-2. Goel Modified Non-Homogeneous Poisson Process
Model (Model FC3)

1. Assumptions

a. Initial fault content:

- . Expected number of software faults to be detected eventually is an unknown fixed quantity.
- . Actual number of faults to be detected is a random variable.

b. Independence of faults:

- . Each failure is caused by one fault and each of them is equally likely to occur
- . Number of software faults detected during non-overlapping testing intervals is independent of others

c. Fault removal process:

- . Fault removal time is negligible
- . No new faults are introduced during the fault removal process

d. Intensity function:

- . Expected number of software faults detected during $(t, t+\Delta t)$ is given by

$$m(t+\Delta t) - m(t) = bct^{c-1}\{a - m(t)\}\Delta t$$

where $m(t)$ = expected number of software faults detected by time t ,

a = expected number of software faults to be eventually detected

b, c = constants.

- . The intensity function or the fault detection rate $\lambda(t)$ is a function of time and is given by

$$\lambda(t) \equiv m'(t) = abct^{c-1}e^{-bt^c}$$

where a, b, c , and $m(t)$ are as defined above.

2. Basic Formulae

Expected number of software faults detected by time t is given by

$$m(t) = a(1 - e^{-bt^c})$$

where a, b , and c are as defined above.

The total number of software faults detected by time t , $N(t)$, under the above assumptions is a NHPP with mean value function $m(t)$ and intensity function $\lambda(t)$ as given above.

The distribution of $N(t)$, hence, is given by

$$P\{N(t) = y\} = \frac{\{m(t)\}^y}{y!} e^{-m(t)}, \quad y = 0, 1, 2, \dots$$

and

$$E\{N(t)\} = m(t) = a(1 - e^{-bt^c})$$

3. Estimation of Parameters

Suppose y_1, y_2, \dots, y_n are the cumulative number of software faults detected by times t_1, t_2, \dots, t_n , respectively.

Then likelihood function for (a, b, c) , given the data pairs

(y_i, t_i) , $i = 1, 2, \dots, n$ is

$$L(a, b, c | y_1, y_2, \dots, y_n, t_1, t_2, \dots, t_n) =$$

$$\Pr\{N(t_1) = y_1, N(t_2) = y_2, \dots, N(t_n) = y_n\}$$

$$= \prod_{i=1}^n \frac{\{m(t_i) - m(t_{i-1})\}^{y_i - y_{i-1}}}{(y_i - y_{i-1})!} e^{-\{m(t_i) - m(t_{i-1})\}}$$

$$= \prod_{i=1}^n \frac{\{a(e^{-bt_{i-1}^c} - e^{-bt_i^c})\}^{y_i - y_{i-1}}}{(y_i - y_{i-1})!} e^{-a(1 - e^{-bt_n^c})}$$

MLE of parameters a, b , and c can be obtained by solving the following non-linear simultaneous equations:

$$y_n = a(1 - e^{-bt_n^c}),$$

$$at_n^c e^{-bt_n^c} = \sum_{i=1}^n \frac{(y_i - y_{i-1})(t_i^c e^{-bt_i^c} - t_{i-1}^c e^{-bt_{i-1}^c})}{(e^{-bt_{i-1}^c} - e^{-bt_i^c})},$$

and

$$at_n^c (\ln t_n) e^{-bt_n^c} = \sum_{i=1}^n \frac{(y_i - y_{i-1})\{t_i^c (\ln t_i) e^{-bt_i^c} - t_{i-1}^c (\ln t_{i-1}) e^{-bt_{i-1}^c}\}}{(e^{-bt_{i-1}^c} - e^{-bt_i^c})}$$

The set of simultaneous equations can be solved numerically for a, b , and c . The solution will be the required maximum likelihood estimates \hat{a} , \hat{b} , and \hat{c} of a , b , and c , respectively.

4. Performance Measures

- Expected number of software faults detected by time t is given by

$$\hat{m}(t) = \hat{a}(1 - e^{-\hat{b}t^{\hat{c}}})$$

- Expected number of remaining faults in the software system at time t is given by

$$E[\hat{N}(t)] = \hat{a}e^{-\hat{b}t^{\hat{c}}}$$

where $\tilde{N}(t)$ = number of faults remaining in the s/w system at time t.

. Software reliability

Let X_k be the time between failures (k-1) and k, and S_k be the time to k failures. Then the conditional reliability function of X_k , given by $S_{k-1} = s$, is

$$\hat{R}_{X_k|S_{k-1}}(x|s) = \exp[-\hat{a}\{e^{-bs^c} - e^{-b(s+x)^c}\}]$$

D-3. Musa Execution Time Model (Model FC4)

1. Assumptions

a. Initial Fault Content:

- . The number of faults in the program (existing before the test phase) is an unknown fixed quantity N .

b. Independence of the Faults:

- . Faults in the program are independent of each other and each fault has a constant average occurrence rate. Failure intervals are independent of each other.

c. Testing Environment:

- . Test space for the program covers the use space.
- . The set of inputs for each run of the program, whether during a test or operational phase, is selected randomly.

d. Fault Removal Process:

- . New faults can be introduced during the correction process.
- . More than one fault can be corrected during each correction time.
- . Time to correct faults is negligible.

e. Hazard Function:

- . Execution times between failures are piecewise exponentially distributed.

Hazard rate is proportional to the expected value of the number of faults remaining, i.e.,

$$Z(\tau) = Kf(N - n_0)$$

where, Z = hazard function

τ = execution time utilized in executing program up to the present

N = initial fault content (existing before the test phase)

n_c = number of faults corrected

f = linear execution frequency (average instruction execution rate divided by number of instructions in the program)

K = proportionality constant, which is an fault exposure ratio which relates fault exposure frequency to linear execution frequency.

2. Basic Formulae

1. Fault correction rate: $\frac{dn_c}{d\tau} = BCZ(\tau)$

where, B = average ratio of the rate of reduction of faults to the rate of failure occurrence (fault reduction factor).

C = average ratio of rate of detection of failures during test to that during use (testing compression factor).

2. $m = \frac{n_c}{B}$ = expected number of failures experienced in correcting n faults

3. $M = \frac{N}{B}$ = expected total number of failures required to expose and remove the N inherent faults

$$4. \frac{dm}{d\tau} + BCfKm = BCfKM$$

$$5. m = M[1 - \exp(-c\tau/MT_0)]$$

where T_0 = initial MTTF before testing.

$$6. n_c = N[1 - \exp(-c\tau/MT_0)]$$

3. Estimation of Parameters:

Likelihood Function

$$L(\tau_1, \tau_2, \dots, \tau_n) = \prod_{i=1}^n \frac{c}{T_0} \left[1 - \frac{i-1}{M}\right] \exp\left[-\tau_i \frac{c}{T_0} \left(1 - \frac{i-1}{M}\right)\right]$$

MLE of parameters T_0 and M can be obtained from the following pair of equations

$$\frac{m}{T_0} - \frac{c}{T_0^2} \sum_{i=1}^m \left[1 - \frac{i-1}{M}\right] \tau_i = 0$$

and

$$\sum_{i=1}^m \frac{1}{M - (i-1)} - \frac{c}{MT_0} \sum_{i=1}^m \tau_i = 0$$

4. Performance Measures:

Reliability: $R(t) = \exp(-\frac{t}{T})$

where

$$T = \text{MTTF} = T_0 \exp(c\tau/MT_0)$$

D-4. Shooman Exponential Model (Model FC5)

1. Assumptions

- a. Initial Fault Content:
 - . An unknown fixed constant N
- b. Independence of faults
 - . Each error in the program is independent of other errors and each of them is equally likely to occur.
- c. Fault Removal Process:
 - . A detected error is corrected with certainty.
 - . No new errors are introduced during debugging
 - . Correction time of an error is negligible.
- d. Hazard Function
 - . Error occurrence rate or correction rate is proportional to the number of remaining errors $n_r(\tau)$, i.e.

$$\begin{aligned}\lambda(t) &= K n_r(\tau) \\ &= K \left[\frac{N}{I} - n_c(\tau) \right]\end{aligned}$$

where,

I = total number of instructions in the program

τ = debugging time since start of system integration

$n_c(\tau)$ = total number of faults corrected during debugging interval τ , normalized with respect to I.

K = proportionality constant

t = operating time of the system measured from its initial activation.

2. Basic Formulae:

Probability that a s/w failure will occur in time interval $(t, t+\Delta t)$ after t hours of successful operation is proportional to the failure rate (hazard function) $Z(t)$, i.e.

$$\Pr\{t < t_f \leq t + \Delta t | t_f > t\} = Z(t)\Delta t = K n_r(t)\Delta t$$

where t_f = operating time to failure

3. Estimation of Parameters:

Parameters N and K can be estimated by running a functional test after two different debugging times, $\tau_1 < \tau_2$ chosen so that $n_c(\tau_1) < n_c(\tau_2)$. Then

$$MTTF_1 = \frac{1}{\lambda_{s1}} = \left[\frac{X_{s1}}{H_1} \right]^{-1} = [K[\frac{N}{I} - n_c(\tau_1)]]^{-1}$$

and

$$MTTF_2 = \frac{1}{\lambda_{s2}} = \left[\frac{X_{s2}}{H_2} \right]^{-1} = [K[\frac{N}{I} - n_c(\tau_2)]]^{-1}$$

where, λ_s = software failure rate

H = total number of run hours (successful).

From the above two equations we get

$$\hat{N} = \frac{I[(\lambda_{s2}/\lambda_{s1})n_c(\tau_1) - n_c(\tau_2)]}{(\lambda_{s2}/\lambda_{s1}) - 1}$$

and hence

$$\hat{K} = \lambda_{s1} / [(\hat{N}/I) - n_c(\tau_1)]$$

4. Performance Measures:

Reliability: $R(t) = \exp[-Z(t)t]$

$$= \exp[-K\{\frac{N}{I} - n_c(\tau)\}t]$$

MTTF: $MTTF = \frac{1}{Z(t)} = \frac{1}{K[\frac{N}{I} - n_c(\tau)]}$

D-5. Geometric Poisson Model (Model FC6)

1. Assumptions

a. Initial Fault Content:

- . No specific value needs to be assumed

b. Independence of the Faults

- . Each fault in the s/w is independent of others and each of them is equally likely to occur.

c. Error Removal Process:

- . Number of faults detected in any debugging interval (fixed) are removed with certainty at the end of the debugging interval.
- . No new fault is introduced during the correction time.
- . Time to correct the detected faults is negligible.

d. Hazard Function:

- . Debugging intervals are fixed.
- . The number of faults occurring in the i -th interval is governed by a Poisson distribution with parameter λK^{i-1} , i.e.

$$Z(t_i) = \lambda K^{i-1},$$

where,

λ = average number of faults occurring
in the first interval

K = a positive number less than 1.

2. Basic Formulae:

The distribution of the number of faults, N_i , detected during i th debugging interval is Poisson with parameter λK^{i-1} , i.e.,

$$\Pr(N_i = n_i) = \frac{(\lambda K^{i-1})^{n_i} \exp[-\lambda K^{i-1}]}{n_i!}$$

$$E(N_i) = \lambda K^{i-1}$$

3. Estimation of Parameters

Likelihood Function:

$$L(n_1, n_2, \dots, n_m) = \Pr(N_1 = n_1, N_2 = n_2, \dots, N_m = n_m)$$

$$= \prod_{i=1}^m \frac{(\lambda K^{i-1})^{n_i} \exp[-\lambda K^{i-1}]}{n_i!}$$

Maximum likelihood estimators of parameters λ and K can be obtained by solving the following pair of equations simultaneously

$$\frac{1}{\lambda} \sum_{i=1}^m n_i - \sum_{i=0}^{m-1} K^i = 0$$

$$\lambda \sum_{i=0}^{m-1} i K^i = \sum_{i=0}^{m-1} i n_{i+1}$$

These two equations yield

$$\frac{(1 - K^m)(1 - K)}{K + (m-1)K^{m+1} - mK^m} = \frac{\sum_{i=1}^m n_i}{\sum_{i=0}^{m-1} i n_{i+1}}$$

from which we can find \hat{K} and hence

$$\hat{\lambda} = \frac{\sum_{i=0}^{m-1} i n_{i+1}}{\sum_{i=0}^{m-1} i \hat{K}^i}$$

4. Performance Measures:

After the i th debugging interval the failure rate is λK^i , therefore

$$\text{Reliability: } R(t) = \Pr\{\text{no failures in } (\sum_{m=1}^i t_m, \sum_{m=1}^i t_m + t]\}$$

$$= e^{-\lambda K^i t}$$

$$\text{MTTF after } i\text{-th debugging interval} = \int_0^{\infty} R(t) dt$$

$$= \frac{1}{\lambda K^i}$$

D-6. Modified Jelinski-Moranda Model (Model FC7)

1. Assumptions

- a. Initial fault content:
 - . An unknown fixed quantity N
- b. Independence of faults:
 - . Each fault in the program is independent of other faults and each of them is equally likely to cause a failure during testing.
 - . Number of faults discovered in any testing interval is independent of that in any other intervals.
- c. Fault removal process:
 - . All detected faults are removed with certainty at the end of each testing interval
 - . Fault removal time is negligible
 - . No new faults are introduced during the fault removal process.
- d. Hazard function
 - . The fault occurrence rate or the hazard function during a testing interval is proportional to the number of remaining faults at the beginning of this interval and for the i th testing interval is given by

$$Z(t_i) = \phi[N - M_{i-1}]$$

where

$$M_j = \sum_{i=1}^j N_i = \text{total number of faults removed up to the end of the } j\text{th testing interval.}$$

t_i = ith testing interval

ϕ = proportionality constant

2. Basic Formulae

Number of faults, N_i , detected during the ith testing interval t_i has a Poisson distribution with rate $\phi[N - M_{i-1}]$ i.e.

$$\Pr\{N_i = n_i\} = \frac{\{\phi(N - M_{i-1})t_i\}^{n_i} e^{-\phi(N - M_{i-1})t_i}}{n_i!}$$

and

$$E[N_i] = \phi(N - M_{i-1})t_i$$

3. Estimation of Parameters

Likelihood Function

$$\begin{aligned} L(n_1, n_2, \dots, n_m) &= \Pr\{N_1 = n_1, N_2 = n_2, \dots, N_m = n_m\} \\ &= \prod_{i=1}^m \frac{\{\phi(N - M_{i-1})t_i\}^{n_i}}{n_i!} \exp[-\phi(N - M_{i-1})t_i] \end{aligned}$$

MLE of parameters N, ϕ can be obtained by solving the following pair of equations simultaneously

$$\sum_{i=1}^m \frac{n_i}{N - M_{i-1}} - \phi \sum_{i=1}^m t_i = 0$$

and

$$\frac{1}{\phi} \sum_{i=1}^m n_i - \sum_{i=1}^m (N - M_{i-1})t_i = 0$$

The above pair of equations yield

$$\sum_{i=1}^m \frac{n_i}{N - M_{i-1}} = \frac{(\sum_{i=1}^m n_i)(\sum_{i=1}^m t_i)}{\sum_{i=1}^m (N - M_{i-1}) t_i}$$

from which we can find \hat{N} and therefore

$$\hat{\phi} = \frac{\sum_{i=1}^m n_i}{\sum_{i=1}^m (\hat{N} - M_{i-1}) t_i}$$

4. Performance Measures:

After the i -th debugging interval the failure rate is $\phi[N - M_i]$

Therefore

$$\begin{aligned} \text{Reliability: } R(t) &= \Pr\{\text{no failure in } (\sum_{m=1}^i t_m, \sum_{m=1}^i t_m + t)\} \\ &= e^{-\phi(N - M_i)t} \end{aligned}$$

MTTF after i -th debugging interval

$$= \int_0^{\infty} R(t) dt = \frac{1}{\phi[N - M_i]}$$

D-7 Modified Geometric De-Eutrophication Model (Model FC8)

1. Assumptions

a. Initial Fault Content:

- . No specific value needs to be assumed

b. Independence of faults:

- . Each fault in the program is independent of other faults and each of them is equally likely to cause a failure during testing.
- . Number of faults discovered in any testing interval is independent of that in any other intervals.

c. Fault removal process:

- . All detected faults are removed with certainty at the end of each testing interval.
- . Fault removal time is negligible
- . No new faults are introduced during the fault removal process.

d. Hazard function

- . The fault occurrence rate or the hazard function during a testing interval is constant but the value changes at the beginning of the next testing interval. For the i th testing interval, t_i , the hazard function is given by

$$Z(t_i) = DK^{M_{i-1}}$$

where, D = fault detection rate during the first testing interval t_1

K = a positive constant less than 1

M_{i-1} = cumulative number of faults detected
up to the end of (i-1)st testing interval.

2. Basic Formulae

Number of faults N_i detected in the i th testing interval t_i has a Poisson distribution with rate $DK^{M_{i-1}}$, i.e.

$$\Pr\{N_i = n_i\} = \frac{(DK^{M_{i-1}}t_i)^{n_i} \exp[-DK^{M_{i-1}}t_i]}{n_i!}$$

and

$$E[N_i] = DK^{M_{i-1}}t_i$$

3. Estimation of Parameters:

Likelihood function:

$$\begin{aligned} L(n_1, n_2, \dots, n_m) &= P_r\{N_1 = n_1, N_2 = n_2, \dots, N_m = n_m\} \\ &= \prod_{i=1}^m \frac{(DK^{M_{i-1}}t_i)^{n_i} \exp[-DK^{M_{i-1}}t_i]}{n_i!} \end{aligned}$$

MLE of parameters D and K can be obtained by solving the following pair of equations simultaneously

$$\frac{1}{D} \sum_{i=1}^m n_i - \sum_{i=1}^m K^{M_{i-1}}t_i = 0$$

and

$$\frac{1}{K} \sum_{i=1}^m n_i M_{i-1} - D \sum_{i=1}^m M_{i-1} K^{M_{i-1}-1} t_i = 0$$

The above two equations yield

$$\frac{1}{K} \sum_{i=1}^m n_i M_{i-1} = \left(\frac{\sum_{i=1}^m n_i}{\sum_{i=1}^m K^{M_{i-1}}t_i} \right) \sum_{i=1}^m M_{i-1} K^{M_{i-1}-1} t_i$$

from which we can find \hat{K} and hence

$$\hat{D} = \frac{\sum_{i=1}^m n_i}{\sum_{i=1}^m \hat{K}^{M_{i-1}} t_i}$$

Performance Measures:

After i -th debugging interval the failure rate is $\frac{M_i}{DK}$

$$\begin{aligned} R(t) &= \Pr\{\text{No failures in } (\sum_{i=1}^i t_i, \sum_{i=1}^i t_i + t]\} \\ &= e^{-DK^{M_i} t} \end{aligned}$$

$$MTTF = \int_0^{\infty} R(t) dt = \frac{1}{\frac{M_i}{DK}} = \text{Mean time to failure after } i\text{-th debugging interval.}$$

D-8 Modified Schick-Wolverton Model (Model FC9)

1. Assumptions

- a. Initial fault content:
 - . An unknown fixed quantity N
- b. Independence of faults:
 - . Each fault in the program is independent of other faults and each of them is equally likely to cause a failure during testing.
 - . Number of faults discovered in any testing interval is independent of that in any other intervals.
- c. Fault removal process:
 - . All detected faults are removed with certainty at the end of each testing interval.
 - . Fault removal time is negligible
 - . No new faults are introduced during the fault removal process.
- d. Hazard function
 - . The fault occurrence rate or the hazard function during a testing interval is proportional to the number of remaining faults at the beginning of this interval and to the total time previously spent in testing (including an "averaged" error search time during the current testing interval). Specifically, the hazard function during the i th testing interval is given by

$$Z(t_i) = \phi[N - M_{i-1}][T_{i-1} + t_i/2]$$

where $M_j = \sum_{i=1}^j N_i$ = total number of faults removed up to the end of the j-th interval

t_i = i-th debugging interval

T_{i-1} = cumulative test time through (i-1)th interval = $\sum_{j=1}^{i-1} t_j$; $T_0 = 0$

ϕ = proportionality constant.

2. Basic Formulae

Number of faults N_i detected in the i-th testing interval of length t_i has a Poisson distribution with rate

$$\phi[N - M_{i-1}][T_{i-1} + \frac{t_i}{2}] \text{ i.e.}$$

$$\Pr\{N_i = n_i\} = \frac{\{\phi[N - M_{i-1}][T_{i-1} + \frac{t_i}{2}]\}^{n_i} \exp[-\phi(N - M_{i-1})(T_{i-1} + \frac{t_i}{2})t_i]}{n_i!}$$

and

$$E[N_i] = \phi[N - M_{i-1}][T_{i-1} + \frac{t_i}{2}]t_i$$

3. Estimation of Parameters

The number of faults detected in each test interval is observed. Suppose the number observed during m such intervals t_1, t_2, \dots, t_m are n_1, n_2 , and n_m , respectively. Then the likelihood of observing n_1 faults in the first interval, interval, n_2 in the second and so on, is given by

$$L(n_1, n_2, \dots, n_m / t_1, t_2, \dots, t_m) = P_r \{N_1 = n_1, N_2 = n_2, \dots, N_m = n_m\}$$

$$= \prod_{i=1}^m \frac{\{\phi [N - M_{i-1}] [T_{i-1} + \frac{t_i}{2}] t_i\}^{n_i} \exp[-\phi (N - M_{i-1}) (T_{i-1} + \frac{t_i}{2}) t_i]}{n_i!}$$

MLE of parameters N and ϕ can be obtained by solving the following pair of equations simultaneously

$$\frac{1}{\phi} \sum_{i=1}^m n_i - \sum_{i=1}^m (N - M_{i-1}) (T_{i-1} + t_i/2) t_i = 0$$

and

$$\sum_{i=1}^m \frac{n_i}{N - M_{i-1}} - \phi \sum_{i=1}^m (T_{i-1} + t_i/2) t_i = 0$$

The above two equations yield

$$\sum_{i=1}^m \frac{n_i}{N - M_{i-1}} - \frac{\sum_{i=1}^m n_i (\sum_{i=1}^m (T_{i-1} + \frac{t_i}{2}) t_i)}{\sum_{i=1}^m (N - M_{i-1}) (T_{i-1} + t_i/2) t_i} = 0$$

from which we can find \hat{N} . Therefore

$$\hat{\phi} = \frac{\sum_{i=1}^m \frac{n_i}{N - M_{i-1}}}{\sum_{i=1}^m (T_{i-1} + t_i/2) t_i}$$

4. Performance Measures:

After $(i-1)$ th debugging interval the failure rate is

$\phi [N - M_{i-1}] [T_{i-1} + t_i/2]$, therefore

AD-A139 240

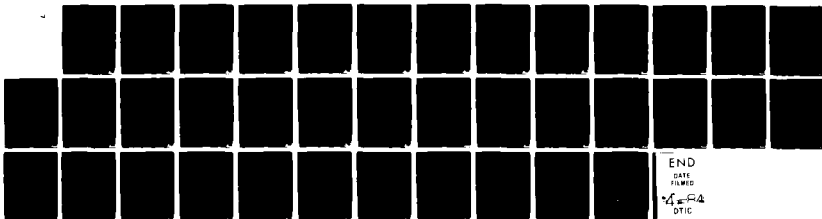
A GUIDEBOOK FOR SOFTWARE RELIABILITY ASSESSMENT(U)
SYRACUSE UNIV NY A L GOEL AUG 83 RADC-TR-83-176
F30602-81-C-0169

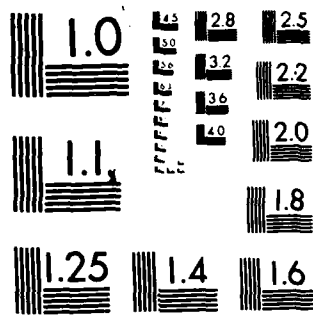
3/3

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

$$\begin{aligned}
\text{Reliability } R(t) &= \Pr\{\text{no failure during } (\sum_{m=1}^{i-1} t_m, \sum_{m=1}^{i-1} t_m + t)\} \\
&= e^{-\phi [N - M_{i-1}] \int_0^t (t_{i-1} + t_i/2) dt_i} \\
&= e^{-\phi [N - M_{i-1}] [T_{i-1} + t/4] t} \\
&= \exp\{-\phi [N - M_{i-1}] [T_{i-1} + \frac{t}{4}] t\}
\end{aligned}$$

$$\text{MTTF after } (i-1)\text{th debugging interval} = \int_0^{\infty} R(t) dt.$$

5. Data Requirements

Data on testing interval lengths t_1, t_2, \dots, t_m and the corresponding number of failures n_1, n_2, \dots, n_m are needed to estimate the parameters N and ϕ .

6. Model Applicability

This model could be used when the testing effort is constant during a given testing interval and if the underlying assumptions are satisfied.

7. Relevant References

The model was suggested in Lipon [LIP74] and used to analyze some failure data by Sukert [SUK76]. Also, the performance of this model was compared in [SUK76]

D-9. Generalized Poisson Model (Model FC10)

1. Assumptions

a. Initial Fault Content:

- . An unknown fixed quantity N

b. Independence of Faults

- . Each fault in the program is independent of the other faults and each of them is equally likely to occur. Number of faults discovered in a testing interval is independent of the number in another interval.

c. Error Removal Process:

- . When faults are removed they are removed with certainty at the ends of the debugging intervals.
- . No new faults are introduced during the correction time.
- . Time to correct faults is negligible.

d. Hazard Function:

- . The number of faults N_i detected in the i th testing interval has a Poisson distribution with mean value function

$$m(t_i) \equiv E[N_i] = \phi[N - M_{i-1}]t_i^\alpha,$$

where,

$$M_j = \sum_{i=1}^j N_i = \text{total number of faults removed up to the end of the } j\text{-th debugging interval.}$$

t_i = i -th debugging interval

ϕ = constant of proportionality

α = constant

i.e.

$$Z(t_i) = \phi \alpha [N - M_{i-1}] t_i^{\alpha-1}$$

2. Basic Formulae:

Number of faults detected, N_i , in the i -th debugging interval t_i has a Poisson distribution, i.e.

$$\Pr\{N_i = n_i\} = \frac{\{\phi(N - M_{i-1})t_i^\alpha\}^{n_i} \exp[-\phi(N - M_{i-1})t_i^\alpha]}{n_i!}$$

and

$$E[N_i] \equiv m(t_i) = \phi(N - M_{i-1})t_i^\alpha$$

3. Estimation of Parameters:

Likelihood Function

$$\begin{aligned} L(n_1, n_2, \dots, n_m) &= \Pr\{N_1 = n_1, N_2 = n_2, \dots, N_m = n_m\} \\ &= \prod_{i=1}^m \frac{\{\phi(N - M_{i-1})t_i^\alpha\}^{n_i} \exp[-\phi(N - M_{i-1})t_i^\alpha]}{n_i!} \end{aligned}$$

MLE of parameters N , ϕ and α can be obtained from the following set of three equations

$$\sum_{i=1}^m \frac{n_i}{N - M_{i-1}} - \phi \sum_{i=1}^m t_i^\alpha = 0$$

$$\frac{1}{\phi} \sum_{i=1}^m n_i - \sum_{i=1}^m (N - M_{i-1}) t_i^\alpha = 0$$

and

$$\sum_{i=1}^m n_i \ln t_i - \phi \sum_{i=1}^m (N - M_{i-1}) t_i^\alpha \ln t_i = 0$$

D-10. IBM Binomial Model (Model FC11)

1. Assumptions

- a. Initial fault content:
 - . The number of faults in the software system (existing before the test phase) is an unknown fixed quantity N .
- b. Independence of faults:
 - . Each failure of the software is caused by one fault and each of them is equally likely to cause a failure during a specified unit interval of testing.
 - . Probability of detecting any one fault during a specified unit interval of testing is constant over all test occasions and independent of other fault detections.
- c. Fault removal process:
 - . Fault removal time is negligible
 - . New faults can be introduced during the fault correction process.
 - . Number of faults introduced on any test occasion is proportional to the number of faults detected.
- d. Testing Process:
 - . The testing phase of the software system (module) is divided into a number of test occasions.
 - . Each test occasion of the software system (module) is further divided into a number of unit intervals.

2. Basic Formulae

a) Binomial module level model

The distribution of the number of faults detected during the i th test occasion in module j follows a binomial distribution with parameters \bar{N}_{ij} and q_{ij} i.e.

$$P\{n_{ij} = x_{ij}\} = \binom{\bar{N}_{ij}}{x_{ij}} q_{ij}^{x_{ij}} (1 - q_{ij})^{\bar{N}_{ij} - x_{ij}}$$

where \bar{N}_{ij} = expected number of faults remaining in module j at the beginning of test occasion i .

n_{ij} = number of faults detected in module j during test occasion i .

q_{ij} = fault detection probability for test occasion i of module j .

$$\bar{N}_{ij} = \omega_j N - \alpha N_{i-1,j}$$

where ω_j = weight assigned to module j (ratio of the size of module j to that of the total system). If all of the modules of the system are under test on occasion i , then

$$\sum_j \omega_j = 1$$

N = total number of faults in the system at the beginning of the first test occasion

$N_{i-1,j}$ = cumulative number of faults detected
in module j through test occasion $i-1$.

α = probability of correcting faults without
introducing additional faults.

$$q_{ij} = [1 - (1-q)^{t_{ij}}]$$

where q = error detection probability during one unit
interval of testing

t_{ij} = total number of unit intervals.

$$E(n_{ij}) = \bar{N}_{ij} q_{ij}$$

b) Binomial system level model

The distribution of the number of faults detected
during the i th test occasion of the software system
follows a binomial distribution with parameters \bar{N}_i and
 q_i , i.e.

$$P\{n_i = x_i\} = \binom{\bar{N}_i}{x_i} q_i^{x_i} (1 - q_i)^{\bar{N}_i - x_i}$$

where \bar{N}_i = expected number of faults remaining in the
system at the beginning of test occasion i .

n_i = number of faults detected in the system
during test occasion i .

q_i = fault detection probability during i th test
occasion of the system.

$$\bar{N}_i = \sum_{j \in J_i} \bar{N}_{ij} = \sum_{j \in J_i} (\omega_j N - \alpha N_{i-1,j})$$

where J_i = set of modules tested on occasion i .

$$q_i = 1 - (1-q)^{t_i}$$

where $t_i = \sum_{j \in J_i} t_{ij}$ = system test effort on occasion i
(total number of unit test intervals for all of the modules tested on occasion i)

$$E[n_i] = \bar{N}_i q_i$$

3. Estimation of Parameters

a) Binomial module level model:

Suppose $n_{11}, n_{12}, \dots, n_{kj}$ are the number of software faults detected in J modules on k test occasions. Then the likelihood function of N , q and α given the actual number of observed faults n_{ij} is

$$\begin{aligned} L(N, q, \alpha | n_{ij}; i = 1, 2, \dots, k; j = 1, 2, \dots, J) \\ = \prod_{i=1}^k \prod_{j=1}^J \binom{\bar{N}_{ij}}{n_{ij}} q_{ij}^{n_{ij}} (1 - q_{ij})^{\bar{N}_{ij} - n_{ij}} \end{aligned}$$

MLE of parameters N , q and α can be obtained by solving the following three equations simultaneously.

$$\frac{\partial \ln L}{\partial N} \equiv 0 = \sum_{i=1}^k \sum_{j=1}^J [w_j \ln \frac{N_{ij}}{\bar{N}_{ij} - n_{ij}} + w_j t_{ij} \ln(1-q)]$$

$$\frac{\partial \ln L}{\partial q} \equiv 0 = \sum_{i=1}^k \sum_{j=1}^J \left[\frac{n_{ij} t_{ij}}{1 - (1-q)^{t_{ij}}} - t_{ij} \bar{N}_{ij} \right]$$

$$\frac{\partial \ln L}{\partial \alpha} \equiv 0 = \sum_{i=1}^k \sum_{j=1}^J \left[N_{i-1,j} \ln \frac{\bar{N}_{ij}}{\bar{N}_{ij} - n_{ij}} + N_{i-1,j} t_{ij} \ln(1-\alpha) \right]$$

b) Binomial system level model:

Suppose n_1, n_2, \dots, n_k are the numbers of software faults detected in the software system on k test occasions ($n_i = \sum_{j=1}^J n_{ij}$). Then the likelihood function of N, q and α , given the actual number of observed faults n_i , is

$$L(N, q, \alpha | n_i; i = 1, 2, \dots, k)$$

$$= \prod_{i=1}^k \binom{\bar{N}_i}{n_i} q_i^{n_i} (1-q_i)^{\bar{N}_i - n_i}$$

MLE of parameters N, q and α can be obtained by solving the following three equations simultaneously.

$$\frac{\partial \ln L}{\partial N} = \sum_{i=1}^k \ln \left(\frac{\bar{N}_i}{\bar{N}_i - n_i} \right) + t_i \ln(1-\alpha) \sum_{j \in J_i} w_j = 0$$

$$\frac{\partial \ln L}{\partial q} = \sum_{i=1}^k \left[\frac{n_i t_i}{1 - (1-q)^{t_i}} + t_i \bar{N}_i \right] = 0$$

$$\frac{\partial \ln L}{\partial \alpha} = \sum_{i=1}^k \left[\ln \frac{\bar{N}_i}{\bar{N}_i - n_i} + t_i \ln(q) \right] \sum_{j \in J_i} N_{i-1,j} = 0$$

4. Performance Measures:

- . Expected number of software faults detected during test occasion i is given by

$$E[n_{ij}] = \bar{N}_{in} \cdot q_{ij} \quad (\text{module level model})$$

$$E[n_i] = \bar{N}_i q_i \quad (\text{system level model})$$

- . Reliability

Suppose that we wish to evaluate the reliability of the software system under test at the end of k testing occasions. Let t_{k+1} be the time interval for the next test occasion. Then the probability that no fault will occur during the $(k+1)$ st test occasion, i.e., the reliability, is given by

$$\hat{R}(t_{k+1}) = (1 - q)^{t_{k+1}(\bar{N}_{k+1})}$$

5. Data Requirements

The data needed are the number of faults detected during each test occasion (for the system level model) or the number of faults detected during each test occasion in each module under test (for the module level model).

6. Model Applicability

This model can be used during unit testing, integration testing and system testing, where the underlying assumptions are satisfied.

7. Relevant References

The model and the detailed data analyses are given in [BRO80].

D-11. IBM Poisson Model (Model FC12)

1. Assumptions

- a. Initial fault content:
 - . The number of faults in the software system (existing before the test phase) is an unknown fixed quantity N .
- b. Independence of faults:
 - . Each failure of the software is caused by one fault and each of them is equally likely to cause a failure during a specified unit interval of testing.
 - . Probability (proportionality factor ϕ) of detecting any one fault during a specified unit interval of testing, is constant over all test occasions and independent of other fault detections.
- c. Fault removal process:
 - . Fault removal time is negligible
 - . New faults can be introduced during the fault correction process.
 - . Number of faults introduced on any test occasion is proportional to the number of faults detected.
- d. Testing process:
 - . The testing phase of the software system (module) is divided into a number of test occasions.
 - . Each test occasion of the software system (module) is further divided into a number of unit intervals.

2. Basic Formulae

a) Poisson module level model

The distribution of the number of faults detected during the i th test occasion in module j follows a Poisson distribution with parameter $\bar{N}_{ij}\phi_{ij}$, i.e.

$$P\{n_{ij} = x_{ij}\} = \frac{e^{-\bar{N}_{ij}\phi_{ij}} (\bar{N}_{ij}\phi_{ij})^{x_{ij}}}{x_{ij}!}$$

where \bar{N}_{ij} = expected number of faults remaining in module j at the beginning of the test occasion i .

n_{ij} = number of faults detected in the module j during test occasion i .

ϕ_{ij} = fault detection rate of each fault in the j th module on test occasion i .

$$\phi_{ij} = 1 - (1-\phi)^{t_{ij}}$$

where ϕ = fault detection rate of each fault during a unit test interval.

t_{ij} = number of unit test intervals for module j on test occasion i .

$$E\{n_{ij}\} = \bar{N}_{ij}\phi_{ij}$$

b) Poisson system level model

The distribution of the number of faults detected during the i th test occasion of the software system

follows a Poisson distribution with parameter $\bar{N}_i \phi_i$

$$P\{n_i = x_i\} = \frac{e^{-\bar{N}_i \phi_i} (\bar{N}_i \phi_i)^{x_i}}{x_i!}$$

where, \bar{N}_i = expected number of faults remaining in the system at the beginning of test occasion i.

n_i = number of faults detected in the system during test occasion i.

ϕ_i = fault detection rate of each fault in the system during test occasion i

$$\phi_i = 1 - (1 - \phi)^{t_i}$$

where, $t_i = \sum_{j \in J_i} t_{ij}$ = system test effort on occasion i.

$$E[n_i] = \bar{N}_i \phi_i$$

3. Estimation of Parameters

a) Poisson module level model:

Suppose $n_{11}, n_{12}, \dots, n_{kJ}$ are the number of software faults detected in J modules on k test occasions. Then the likelihood function of N, ϕ and α given the actual number of observed faults n_{ij} is

$$\begin{aligned} L(N, \phi, \alpha | n_{ij}; i = 1, 2, \dots, k; j = 1, 2, \dots, J) \\ = \prod_{i=1}^k \prod_{j=1}^J \frac{(\bar{N}_{ij} \phi_{ij})^{n_{ij}} e^{-\bar{N}_{ij} \phi_{ij}}}{n_{ij}!} \end{aligned}$$

MLE of parameters N, ϕ and α can be obtained by

solving the following three equations simultaneously.

$$\frac{\partial \ln L}{\partial N} \equiv 0 = \sum_{i=1}^k \sum_{j=1}^J w_j \left[\frac{n_{ij}}{\bar{N}_{ij}} - \phi_{ij} \right]$$

$$\frac{\partial \ln L}{\partial \phi} \equiv 0 = \sum_{i=1}^k \sum_{j=1}^J t_{ij} (1-\phi)^{t_{ij}} \left[\frac{n_{ij}}{1 - (1-\phi)^{t_{ij}}} - \bar{N}_{ij} \right]$$

$$\frac{\partial \ln L}{\partial \alpha} \equiv 0 = \sum_{i=1}^k \sum_{j=1}^J N_{i-1,j} \left[\frac{n_{ij}}{\bar{N}_{ij}} - \phi_{ij} \right]$$

b) Poisson system level model:

Suppose n_1, n_2, \dots, n_k are the numbers of software faults detected in the software system on k test occasions.

Then the likelihood function of N, ϕ and α given the actual number of observed faults n_i is

$$\begin{aligned} L(N, \phi, \alpha \mid n_i; i = 1, 2, \dots, k) \\ = \prod_{i=1}^k \frac{(\bar{N}_i \phi_i)^{n_i} e^{-\bar{N}_i \phi_i}}{n_i!} \end{aligned}$$

MLE of the parameters N, ϕ and α can be obtained by solving the following three equations simultaneously

$$\frac{\partial \ln L}{\partial N} \equiv 0 = \sum_{i=1}^k \left(\sum_{j \in J_i} w_j \right) \left(\frac{n_i}{\bar{N}_i} - \phi_i \right)$$

$$\frac{\partial \ln L}{\partial \phi} \equiv 0 = \sum_{i=1}^k t_i (1-\phi)^{t_i} \left[\frac{n_i}{1 - (1-\phi)^{t_i}} - \bar{N}_i \right]$$

$$-\frac{\partial \ln L}{\partial \alpha} \equiv 0 = \sum_{i=1}^k \left(\sum_{j \in J_i} N_{i-1,j} \right) \left(\frac{n_i}{\bar{N}_i} - \phi_i \right)$$

4. Performance Measures:

- . Expected number of software faults detected during test occasion i is given by

$$E[n_{ij}] = \bar{N}_{ij} \cdot \phi_{ij} \quad (\text{module level model})$$

$$E[n_i] = \bar{N}_i \phi_i \quad (\text{system level model})$$

- . Reliability

Consider that at the end of K testing occasions we wish to evaluate the reliability of the software system under test. Let t_{K+1} be the time interval for the next test occasion, then the probability that no fault will occur during $(K+1)$ th test occasion is given by

$$\hat{R}(t_{K+1}) = e^{-\bar{N}_{K+1} \phi_{K+1} t_{K+1}}$$

$$\text{where } \phi_{K+1} = 1 - (1 - \phi)^{t_{K+1}}$$

5. Data Requirements

The data needed are the number of faults detected during each test occasion (for the system level model) or the number of faults detected during each test occasion in each module under test (for the module level model).

6. Model Applicability

This model can be used during unit testing, integration testing and system testing, where the underlying assumptions are satisfied.

7. Relevant References

The model and the detailed data analyses are given in [BRO80].

APPENDIX E

DETAILS OF COMBINATORIAL MODELS

In this appendix we provide details of the underlying assumptions and other relevant concepts for the fault seeding and input domain based models discussed in Section 5.

E.1 Mills' Hypergeometric Model (Model FS1)

Assumptions

- . Faults are seeded at random. Each portion of the program has the same probability of having a seeded fault.
- . Probability of detecting an indigenous fault is the same as that of detecting a seeded fault.

Basic Formulae

Let

n_s = number of seeded faults

x_s = number of seeded faults detected during testing

n_I = total number of indigenous faults

x_I = number of indigenous faults detected during testing

Then the joint probability of finding x_I indigenous faults and x_s seeded faults in $n_I + n_s$ tests is given by the hyper-

geometric distribution as follows

$$\left\{ \begin{array}{l} x_I \text{ indigenous faults and} \\ x_s \text{ seeded faults in} \\ n_I + n_s \text{ tests} \end{array} \right\} = \frac{\binom{n_I}{x_I} \binom{n_s}{x_s}}{\binom{n_I + n_s}{x_I + x_s}}$$

The maximum likelihood estimate of n_I is given by

$$\hat{n}_I = \frac{n_s \cdot x_I}{x_s}$$

Lipow's Extension [LIP72]

The probability of finding x_I indigeneous faults, x_s seeded faults, and (therefore) $N-x_I-x_s$ tests with no faults found in N statistically independent tests is given by

$$P_N(x_I, x_s; q, n_I, n_s) = \binom{N}{x_I+x_s} q^{x_I+x_s} (1-q)^{N-x_I-x_s} \frac{\binom{n_I}{x_I} \binom{n_s}{x_s}}{\binom{n_I+n_s}{x_I+x_s}} \sum_{i=0}^{n_I+n_s} \binom{N}{i} q^i (1-q)^{N-i}$$

The mle's of n_I and q are given by

$$\hat{q} = \frac{x_I + x_s}{N}$$

and

$$\hat{n}_I = \begin{cases} \lceil \frac{x_I}{x_s} n_s \rceil & \text{if } x_I + x_s \geq 1 \\ 0 & \text{if } x_I + x_s = 0 \\ x_I n_s & \text{if } x_s = 0. \end{cases}$$

Basin's Extension [BAS74]

Basin suggested a method, the so-called two-state edit procedure, where one programmer searches for faults and records n_1 faults out of a total of N unknown indigenous faults. A second programmer edits the program independently to record r faults out of N (unknown) faults. The two lists of faults are then compared. The probability of k

faults in the second programmer being included in the first programmer's list is also given by the hypergeometric distribution, i.e.

$$q_k(N) = \frac{\binom{n_1}{k} \binom{N-n_1}{r-k}}{\binom{N}{r}}$$

The mle of N is then given by

$$\hat{N} = \left\lceil \frac{n_1 r}{k} \right\rceil$$

Appendix F

SELECTED SOFTWARE ENGINEERING TERMS*

Abort:

To terminate a process prior to completion.

Acceptance criteria:

The criteria a software product must meet to successfully complete a test phase or meet delivery requirements.

Acceptance testing:

Formal testing conducted to determine whether a system satisfies its acceptance criteria and to enable the customer to determine whether to accept the system. See also qualification testing, system testing.

Accuracy:

- (1) A quality of that which is free of error. (ISO)
- (2) A qualitative assessment of freedom from error, a high assessment corresponding to a small error. (ISO)
- (3) A quantitative measure of the magnitude of error, preferably expressed as a function of the relative error, a high value of this measure corresponding to a small error. (ISO)
- (4) A quantitative assessment of freedom from error.

Algorithm:

- (1) A finite set of well-defined rules for the solution of a problem in a finite number of steps, e.g., a complete specification of a sequence of arithmetic operations for evaluating $\sin x$ to a given precision. (ISO)
- (2) A finite set of well-defined rules which gives a sequence of operations for performing a specific task.

Analytical model:

A representation of a process or phenomenon by a set of solvable equations. See also simulation.

Application software:

Software specifically produced for the functional use of a computer system, e.g., software for navigation, gun fire control, payroll, general ledger, etc. Contrast with system software.

Assignment statement:

An instruction used to express a sequence of operations, or used to assign operands to specified variables, or symbols, or both. (ANSI)

* Taken from "A Glossary of Software Engineering Terminology" (IEEE Project 729), IEEE Inc., New York 10017.

Automated design tool:

A software tool which aids in the synthesis, analysis, modeling, or documentation of a software design. Examples include simulators, analytic aids, design representation processors, and documentation generators.

Automated test case generator:

See automated test generator.

Automated test data generator:

See automated test generator.

Automated test generator:

A Software tool that accepts as input a computer program and test criteria, generates test input data that meet these criteria, and, sometimes, determines the expected results.

Automated verification system:

A software tool that accepts as input a computer program and a representation of its specification, and produces, possibly with human help, a correctness proof or disproof of the program. See also automated verification tools.

Automated verification tools:

A class of software tools used to evaluate products of the software development process. These tools aid in the verification of such characteristics as correctness, completeness, consistency, traceability, testability, and adherence to standards. Examples include design analyzers, automated verification systems, static analyzers, dynamic analyzers, and standards enforcers.

Availability:

- (1) The probability that software will be able to perform its designated system function when required for use.
- (2) The ratio of system up-time to the total operating time.
- (3) The ability of an item to perform its designated function when required for use. (ANSI/ASQC A3-1978)

Availability model:

A model used for predicting, estimating, or assessing availability.

Baseline:

- (1) A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.
- (2) A configuration identification document or a set of such documents formally designated and fixed at a specific time during a CI's life cycle. Baselines, plus approved changes from those baselines, constitute the current configuration identification. For configuration management there are three baselines, as follows:

- a) Functional baseline. The initial approved functional configuration.
- b) Allocated baseline. The initial approved allocated configuration.
- c) Product baseline. The initial approved or conditionally approved product configuration identification. (DoD-STD 480A)

Bottom-up design:

The design of a system starting with the most basic or primitive components and proceeding to higher level components that use the lower level ones. Contrast with top-down design.

Bug:

See fault.

Bug seeding:

See fault seeding.

Build:

An operational version of a software product incorporating a specified subset of the capabilities that the final product will include.

Certification:

- (1) A written guarantee that a system or computer program complies with its specified requirements.
- (2) A written authorization which states that a computer system is secure and is permitted to operate in a defined environment with or producing sensitive information.
- (3) The formal demonstration of system acceptability to obtain authorization for its operational use.
- (4) The process of confirming that a system, software subsystem, or computer program is capable of satisfying its specified requirements in an operational environment. Certification usually takes place in the field under actual conditions, and is utilized to evaluate not only the software itself, but also the specifications to which the software was constructed. Certification extends the process of verification and validation to an actual or simulated operational environment.
- (5) The procedure and action by a duly authorized body of determining, verifying, and attesting in writing to the qualifications of personnel, processes, procedures or items in accordance with applicable requirements. (ANSI/ASQC A3-1978)

Chief programmer:

The leader of a chief programmer team; a senior-level programmer whose responsibilities include producing key portions of the software assigned to the team, coordinating the activities of the team, reviewing the work of the other team members, and having an overall technical understanding of the software being developed.

Chief programmer team:

A software development group that consists of a chief programmer, a backup programmer, a secretary/librarian and additional programmers and specialists as needed and employs support procedures designed to enhance group communication and make optimum use of each member's skills.

Code:

- (1) A set of unambiguous rules specifying the manner in which data may be represented in a discrete form. (ISO)
- (2) To represent data or a computer program in a symbolic form that can be accepted by a processor. (ISO)
- (3) To write a routine. (ANSI)
- (4) Loosely, one or more computer programs, or part of a computer program.
- (5) The encryption of data for security purposes.

Code generator:

A program or program function, often part of a compiler, which transforms a computer program from some intermediate level of representation (often the output of a parser) into a lower level representation such as assembly code or machine code.

Comment:

- (1) Information embedded within a computer program, command language, or set of data which is intended to provide clarification to human readers and that does not effect machine interpretation.
- (2) A description, reference, or explanation added to or interspersed among the statements of the source language, that has no effect in the target language. (ISO)

Complexity:

The degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, the types of data structures, and other system characteristics.

Computer:

- (1) A functional unit that can perform substantial computation, including numerous arithmetic operations, or logic operations without intervention by a human operator during a run. (ISO)
- (2) A functional programmable unit that consists of one or more associated processing units and peripheral equipment, that is controlled by internally stored programs, and that can perform substantial computation, including numerous arithmetic operations or logic operations, without human intervention.

Computer data:

Data available for communication between or within computer equipment. Such data can be external (in computer-readable form) or resident within the computer equipment and can be in the form of analog or digital signals.

Computer program:

A sequence of instructions suitable for processing by a computer. Processing may include the use of an assembler, a compiler, an interpreter, or a translator to prepare the program for execution as well as to execute it. (ISO) See also program.

Computer program abstract:

A brief description of a computer program, providing sufficient information for potential users to determine the appropriateness of the computer program to their needs and resources.

Computer system:

A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations, and that can execute programs that modify themselves during their execution. A computer system may be a standalone unit or may consist of several interconnected units. Synonymous with ADP system, computing system. (ISO)

Corrective maintenance:

Maintenance performed specifically to overcome existing faults. (ISO)
See also software maintenance.

Correctness:

- (1) The extent to which software is free from design defects and from coding defects; i.e., fault free.
- (2) The extent to which software meets its specified requirements.
- (3) The extent to which software meets user expectations.

Criticality:

A classification of a software error or fault based upon an evaluation of the degree of impact of that error or fault on the development or operation of a system. Often used to determine whether or when a fault will be corrected.

Data:

A representation of facts, concepts or instructions in a formalized manner suitable for communication, interpretation, or processing by human or automatic means. (ISO) See also computer data, error data, software experience data, reliability data.

Debugging:

The process of locating, analyzing, and correcting suspected faults. Compare with testing.

Debugging model:

See error model.

Defect:

See fault.

Definition phase:

See requirements phase.

Design:

- (1) The process of defining the software architecture, components, modules, interfaces, test approach, and data for a software system to satisfy specified requirements.
- (2) The results of the design process.

Development methodology:

A systematic approach to the creation of software that defines development phases and specifies the activities, products, verification procedures, and completion criteria for each phase.

Embedded computer system:

A computer system that is integral to a larger system whose primary purpose is not computational, e.g., a computer system in a weapon, aircraft, command and control, or rapid transit system.

Embedded software:

Software for an embedded computer system.

Error:

- (1) A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. (ANSI)
- (2) Human action which results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, incorrect translation or omission of a requirement in the design specification. This is not a preferred usage.

See also failure, fault.

Error analysis:

- (1) The process of investigating an observed software fault with the purpose of tracing the fault to its source.
- (2) The process of investigating an observed software fault to identify such information as the cause of the fault, the phase of the development process during which the fault was introduced, methods by which the fault could have been prevented or detected earlier, and the method by which the fault was detected.
- (3) The process of investigating software errors, failures, and faults to determine quantitative rates and trends.

Error category:

One of a set of classes into which an error, fault, or failure might fall. Categories may be defined for the cause, criticality, effect, life cycle phase when introduced or detected, or other characteristics of the error, fault, or failure.

Error data:

A term commonly (but not precisely) used to denote information describing software problems, faults, failures and changes, their characteristics, and the conditions under which they are encountered or corrected.

Error model:

A mathematical model used to predict or estimate the number of remaining faults, reliability, required test time or similar characteristics of a software system. See also error prediction.

Error prediction:

A quantitative statement about the expected number or nature of software problems, faults, or failures in a software system. See also error model.

Error prediction model:

See error model.

Error seeding:

See fault seeding.

Execution:

The process of carrying out an instruction or the instructions of a computer program by a computer. (ISO)

Execution time:

- (1) The amount of actual or central processor time used in executing a program.
 - (2) The period of time during which a program is executing.
- See also run time.

Execution time theory:

A theory that uses cumulative execution time as the basis for estimating software reliability.

Executive program:

See supervisory program.

Failure:

- (1) The termination of the ability of a functional unit to perform its required function. (ISO)
- (2) The inability of a system or system component to perform a required function within specified limits. A failure may be produced when a fault is encountered.
- (3) A departure of program operation from program requirements.

Failure category:

See error category.

Failure data:

See error data.

Failure rate:

- (1) The ratio of the number of failures to a given unit of measure, e.g., failures per unit of time, failures per number of transactions, failures per number of computer runs.
- (2) In reliability modeling, the ratio of the number of failures of a given category or severity to a given period of time, e.g., failures per second of execution time, failures per month.

Synonymous with failure ratio.

Failure ratio:

See failure rate.

Failure recovery:

The return of a system to a reliable operating state after failure.

Fault:

- (1) An accidental condition that causes a functional unit to fail to perform its required function. (ISO)
- (2) A manifestation of an error⁽²⁾ in software. A fault, if encountered, may cause a failure.

Synonymous with bug.

Fault category:

See error category.

Fault insertion:

See fault seeding.

Fault seeding:

The process of intentionally adding a known number of faults to those already in a computer program for the purpose of estimating the number of indigenous faults in the program.

Synonymous with bug seeding.

Fault tolerance:

The built-in capability of a system to provide continued correct execution in the presence of a limited number of hardware or software faults.

Formal testing:

The process of conducting testing activities and reporting results in accordance with an approved test plan.

Function:

- (1) A specific purpose of an entity or its characteristic action. (ANSI)
- (2) A subprogram that is invoked during the evaluation of an expression in which its name appears and that returns a value to the point of invocation. Contrast with subroutine.

Hardware:

Physical equipment used in data processing, as opposed to computer programs, procedures, rules, and associated documentation. Contrast with software. (ISO)

Imperfect debugging:

In reliability modeling, the assumption that attempts to correct or remove a detected fault are not always successful.

Indigenous fault:

A fault existing in a computer program that has not been inserted as part of a fault seeding process.

Inspection:

- (1) A formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problems. Contrast with walk-through.
- (2) A phase of quality control which by means of examination, observation or measurement determines the conformance of materials, supplies, components, parts, appurtenances, systems, processes or structures to predetermined quality requirements. (ANSI 45.2.10-1973)

Installation and checkout phase:

The period of time in the software life cycle during which a software product is integrated into its operational environment and tested in this environment to ensure that it performs as required.

Integration:

The process of combining software elements, hardware elements, or both into an overall system.

Integration testing:

An orderly progression of testing in which software elements, hardware elements, or both are combined and tested, until the entire system has been integrated. See also system testing.

Interface testing:

Testing conducted to ensure that program of system components pass information or control correctly to one another.

Life cycle:

See software life cycle.

Maintainability:

- (1) The ease with which software can be maintained.
- (2) The ease with which maintenance of a functional unit can be performed in accordance with prescribed requirements. (ISO)
- (3) Ability of an item under stated conditions of use to be retained in, or restored to, within a given period of time, a specified state in which it can perform its required functions when maintenance is performed under stated conditions and while using prescribed procedures and resources. (ANSI/ASQC A3-1978)

Maintenance:

See software maintenance.

Maintenance phase:

See operation and maintenance phase.

Maintenance plan:

A document that identifies the management and technical approach that will be used to maintain software products. Typically included are topics such as tools, resources, facilities, and schedules.

Model:

A representation of a real world process, device, or concept. See also analytical model, availability model, debugging model, error model, reliability model, simulation, statistical test model.

Module:

- (1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading, e.g., the input to, or output from, an assembler, compiler, linkage editor, or executive routine. (ANSI)
- (2) A logically separable part of a program.

Mutation:

See program mutation.

Object program:

A fully compiled or assembled program that is ready to be loaded into the computer. (ISO) Contrast with source program.

Operating system:

Software that controls the execution of programs. An operating system may provide services such as resource allocation scheduling, input/output control, and data management. Although operating systems are predominantly software, partial or complete hardware implementations are possible. (ISO) An operating system provides support in a single spot rather than forcing each program to be concerned with controlling hardware. See also system software.

Operation and maintenance phase:

The period of time in the software life cycle during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements.

Operational:

Pertaining to the status given a software product once it has entered the operation and maintenance phase.

Operational reliability:

The reliability of a system or software subsystem in its actual use environment. Operational reliability may differ considerably from reliability in the specified or test environment.

Operational testing:

Testing performed by the end user on software in its normal operating environment. (DOD usage)

Parameter:

- (1) A variable that is given a constant value for a specified application and that may denote the application. (ISO)
- (2) A variable that is used to pass values between program routines. See also actual parameter, formal parameter.

Perfective maintenance

Maintenance performed to improve performance, maintainability, or other software attributes. See also adaptive maintenance, corrective maintenance.

Performance:

- (1) The ability of a computer system or subsystem to perform its functions.
- (2) A measure of the ability of a computer system or subsystem to perform its functions, e.g., response time, throughput, number of transactions. See also performance requirement.

Performance evaluation:

The technical assessment of a system or system component to determine how effectively operating objectives have been achieved.

Performance requirement:

A requirement that specifies a performance characteristic that a system or system component must possess, e.g., speed, accuracy, frequency.

Performance specification:

- (1) A specification that sets forth the performance requirements for a system or system component.
- (2) Synonymous with requirements specification. (U.S. Navy usage)

Program:

- (1) A computer program.
- (2) A schedule or plan that specifies actions to be taken.
- (3) To design, write, and test computer programs. (ISO)

Program correctness:

See correctness.

Program extension:

An enhancement made to existing software to increase the scope of its capabilities.

Program mutation:

- (1) A program version purposely altered from the intended version to evaluate the ability of program test cases to detect the alteration. Synonymous with program mutant.
- (2) The process of creating program mutations in order to evaluate the adequacy of program test data.

Program validation:

Synonymous with computer program validation. See validation.

Proof of correctness:

- (1) A formal technique used to prove mathematically that a program satisfies its specifications. See also partial correctness, total correctness.
- (2) A program proof that results from applying this technique.

Qualification testing:

Formal testing, usually conducted by the developer for the customer, to demonstrate that the software meets its specified requirements. See also acceptance testing, system testing.

Quality:

- (1) The totality of features and characteristics of a product or service that bear on its ability to satisfy given needs (ANSI. ASQC A3-1978).
- (2) See software quality.

Quality assurance:

A planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements. (IEEE Standard 730)

Quality metric:

A quantitative measure of the degree to which software possesses a given attribute which affects its quality.

Regression testing:

Selective retesting to detect faults introduced during modification of a system or system component to verify that modifications have not caused unintended adverse effects, or to verify that a modified system or system component still meets its specified requirements.

Reliability:

- (1) The ability of an item to perform a required function under stated conditions for a stated period of time (ANSI/ASQC A3-1978 and IEC 271-1974)
- (2) See software reliability.

Reliability, numerical:

The probability that an item will perform a required function under stated conditions for a stated period of time. (ANSI/ASQC A3-1978)

Reliability assessment:

The process of determining the achieved level of reliability of an existing system or system component.

Reliability data:

Information necessary to assess the reliability of software at selected points in the software life cycle. Examples include error data and time data for reliability models, program attributes such as complexity, and programming characteristics such as development techniques employed and programmer experience.

Reliability evaluation:

See reliability assessment

Reliability growth:

The improvement in software reliability which results from correcting faults in the software.

Reliability model:

A model used for predicting, estimating, or assessing reliability.
See also reliability assessment.

Requirement:

- (1) A condition or capability needed by a user to solve a problem or achieve an objective.
- (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. The set of all requirements forms the basis for subsequent development of the system or system component. See also requirements analysis, requirements phase, requirements specification.

Retirement phase:

The period of time in the software life cycle during which support for a software product is terminated.

Robustness:

The extent to which software can continue to operate correctly despite the introduction of invalid inputs.

Run time:

- (1) A measure of the time expended to execute a program. While it ordinarily reflects the expended central processor time, it may also include peripheral processing and peripheral accessing time, e.g., a run time of 5 hours.
- (2) The instant at which a program begins to execute.
See also execution time.

Seeding:

See fault seeding.

Severity:

See criticality.

Software:

- (1) Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system. See also application software, system software. Contrast with hardware.
- (2) Programs, procedures, rules, and any associated documentation pertaining to the operation of a computer system. (ISO)

Software data base:

A centralized file of data definitions and present values for data common to, and located internal to, an operational software system.

Software development cycle:

- (1) The period of time that begins with the decision to develop a software product and ends when the product is delivered. This cycle typically includes a requirements phase, design phase, implementation phase, test phase, and sometimes, installation and checkout phase. Contrast with software life cycle.
- (2) The period of time that begins with the decision to develop a software product and ends when the product is no longer being enhanced by the developer.
- (3) Sometimes used as a synonym for software life cycle.

Software development process:

The process by which user needs are translated into software requirements, software requirements are transformed into design, the design is implemented in code, and the code is tested, documented, and certified for operational use.

Software documentation:

Technical data or information, including computer listings and printouts, in human-readable form, that describe or specify the design or details, explain the capabilities, or provide operating instructions for using the software to obtain desired results from a software system. See also documentation, system documentation, user documentation.

Software engineering:

The systematic approach to the development, operation, maintenance and retirement of software.

Software experience data:

Data relating to the development or use of software that could be useful in developing models, reliability predictions, or other quantitative descriptions of software.

Software life cycle:

The period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life cycle typically includes a requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. Contrast with software development cycle.

Software maintenance:

- (1) Modification of a software product after delivery to correct faults.
- (2) Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment. See also adaptive maintenance, corrective maintenance, perfective maintenance.

Software product:

A software entity designated for delivery to a user.

Software quality:

- (1) The totality of features and characteristics of a software product that bears on its ability to satisfy given needs; e.g., conform to specifications.
- (2) The degree to which software possesses a desired combination of attributes.
- (3) The degree to which a customer or user perceives that software meets his or her composite expectations.
- (4) The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.

Software quality assurance:

See quality assurance.

Software reliability:

- (1) The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered.
- (2) The ability of a program to perform a required function under stated conditions for a stated period of time.

Software tool:

A computer program used to help develop, test, analyze, or maintain another computer program or its documentation, e.g., automated design tool, compiler, test tool, maintenance tool.

Source program:

- (1) A computer program that must be compiled, assembled, or interpreted before being executed by a computer.
- (2) A computer program expressed in a source language. Contrast with object program. (ISO)

Specification verification:

See verification.

Statistical test model:

A model that relates program faults to the input data set (or sets) which cause them to be encountered. The model also gives the probability that these faults will cause the program to fail.

Structured design:

A disciplined approach to software design which adheres to a specified set of rules based on principles such as top-down design, stepwise refinement, and data flow analysis.

Structured program:

A program constructed of a basic set of control structures, each one having one entry point and one exit. The set of control structures typically includes: sequence of two or more instructions, conditional selection of one of two or more instructions or sequences of instructions, and repetition of an instruction or a sequence of instructions.

Subprogram:

A program unit that may be invoked by one or more other program units. Examples are procedure, function, subroutine.

Symbolic execution:

A verification technique in which program execution is simulated using symbols rather than actual values for input data, and program outputs are expressed as logical or mathematical expressions involving these symbols.

System reliability:

The probability that a system, including all hardware and software subsystems, will perform a required task or mission for a specified time in a specified environment. See also operational reliability, software reliability.

System software:

Software designed for a specific computer system or family of computer systems to facilitate the operation and maintenance of the computer system and associated programs, e.g. operating systems, compilers, utilities. Contrast with application software.

System testing:

The process of testing an integrated hardware and software system to verify that the system meets its specified requirements. See also acceptance testing, qualification testing.

Test case:

A specific set of test data and associated procedures developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. See also testing.

Test driver:

A driver that invokes the item under test and may provide test inputs and report test results.

Test log:

A chronological record of all relevant details of a testing activity.

Test phase:

The period of time in the software life cycle during which the components of a software product are evaluated and integrated, and the software product is evaluated to determine whether or not requirements have been satisfied.

Test plan:

A document prescribing the approach to be taken for intended testing activities. The plan typically identifies the items to be tested, the testing to be performed, test schedules, personnel requirements, reporting requirements, evaluation criteria, and any risks requiring contingency planning.

Test procedure:

Detailed instructions for the set up, operation, and evaluation of results for a given test. A set of associated procedures is often combined to form a test procedures document.

Test validity:

The degree to which a test accomplishes its specified goal.

Testability:

- (1) The extent to which software facilitates both the establishment of test criteria and the evaluation of software with respect to those criteria.
- (2) The extent to which the definition of requirements facilitates analysis of the requirements to establish test criteria.

Testing:

The process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results. Compare with debugging.

Tolerance:

The ability of a system to provide continuity of operation under various abnormal conditions.

Total correctness:

In proof of correctness, a designation indicating that a program's output assertions follow logically from its input assertions and processing steps, and that, in addition, the program terminates under all specified input conditions. Contrast with partial correctness.

Utility software:

Computer programs or routines designed to perform some general support function required by other application software, by the operating system, or by system users.

Validation:

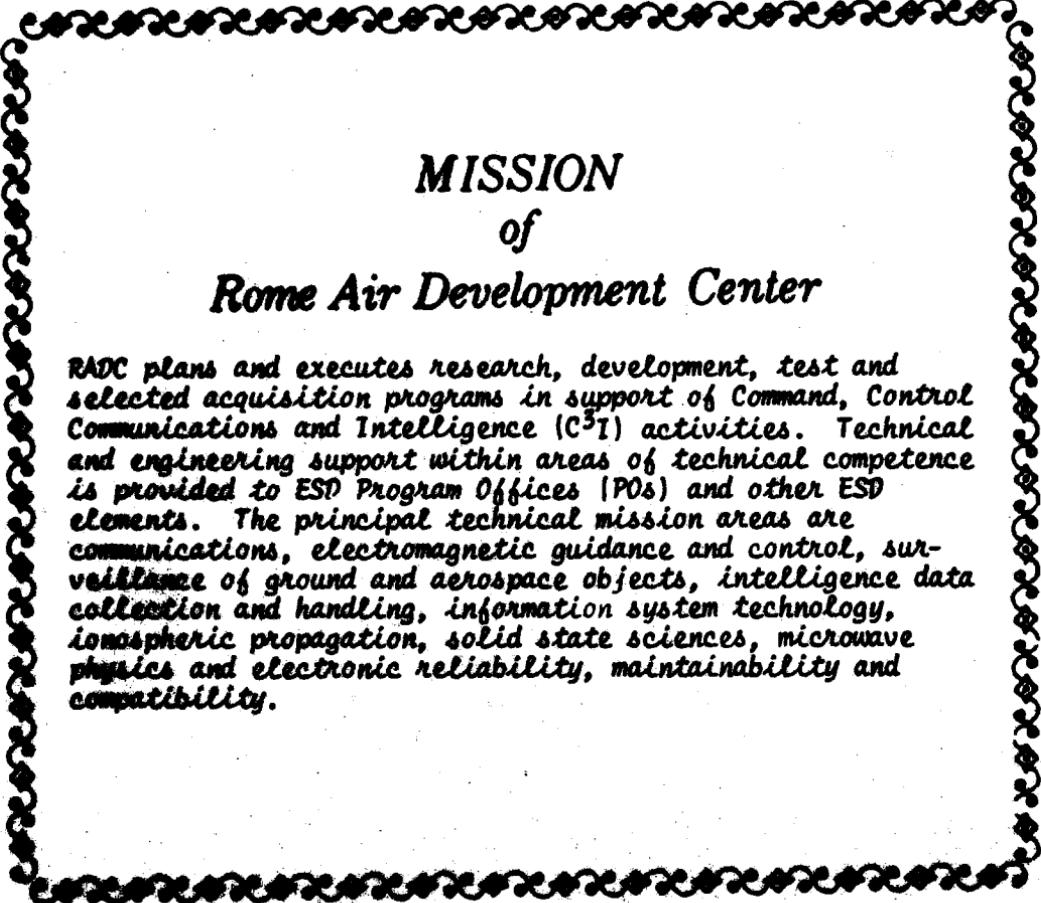
The process of evaluating software at the end of the software development process to ensure compliance with software requirements. See also verification.

Verification:

- (1) The process of determining whether the products of a given phase of the software development cycle fulfill the requirements established during the previous phase. See also validation.
- (2) Formal proof of program correctness. See proof of correctness.
- (3) The act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether items, processes, services, or documents conform to specified requirements. (ANSI/ASQC A3-1978).

Walk-through:

A review process in which a designer or programmer leads one or more other members of the development team through a segment of design or code that he or she has written, while the other members ask questions and make comments about technique, style, possible errors, violation of development standards, and other problems. Contrast with inspection.



MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C³I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

END

DATE
FILMED

4-84

DTIC